The Common Order-Theoretic Structure of Version Spaces and ATMS's

Carl A. Gunter
University of Pennsylvania

Teow-Hin Ngair
National University of Singapore

Devika Subramanian
Rice University

Abstract

We demonstrate how order-theoretic abstractions can be useful in identifying, formalizing, and exploiting relationships between seemingly dissimilar AI algorithms that perform computations on partially-ordered sets. In particular, we show how the order-theoretic concept of an *anti-chain* can be used to provide an efficient representation for such sets when they satisfy certain special properties. We use anti-chains to identify and analyze the basic operations and representation optimizations in the version space learning algorithm [10] and the assumption-based truth maintenance system (ATMS) [2, 3]. Our analysis allows us to (1) extend the known theory [7, 10, 8] of admissibility of concept spaces for incremental version space merging, and (2) develop new, simpler label-update algorithms for ATMS's with DNF assumption formulas.

Contents

1 Introduction

This paper shows how the order-theoretic concept of an *anti-chain* provides a useful abstraction for the representation of partial information. The primary contribution is the isolation of a collection of eight primitive operations on anti-chains and a demonstration of how these operations can be used in some circumstances where the efficient representation of partial information is a key concern. We call this set of operations the *anti-chain algebra.* The description and basic properties of the operations are given in Section 2; the remainder of the paper is devoted to applications of the anti-chain algebra. The discovery of a useful algebra of expressions can be a powerful technique. For example, Codd's introduction of an algebra of relations aided the development of practical and semantically clear database query languages. While it is ambitious to think that an anti-chain algebra will do for knowledge representation what Codd's relational algebra did for query languages, significant insights can be obtained from recognizing when anti-chains are a good representation.

To show how the anti-chain algebra can be useful in expressing representations for partial information, we examine two well-known approaches to manipulating and refining partial information. The first of these, the *version space* (VS) algorithm, is used for inductive learning based on forming concept descriptions from examples. This is the topic of Section 3. The second, *Assumption-Based Truth Maintenance System* (ATMS) algorithm, records dependencies between propositions by maintaining all of the support sets for a proposition. Our discussion of ATMS's is broken into two parts: Section 4 studies the 'basic' ATMS, which uses Horn clauses for its base of facts, while Section 5 studies the 'extended' ATMS, which extends the basic ATMS by permitting the use of facts in disjunctive normal form in addition to Horn clauses. Although only a cursory knowledge of the VS, ATMS, and extended ATMS representations is required to see that the ideas they embody have many things in common, appropriate mathematical structures are needed to obtain an account of this commonality that is rigorous enough to show how the methods can share notations, facts supporting correctness proofs, optimizations, and even code modules. We show that the anti-chain algebra achieves this. An appendix describing interfaces for modules implementing our anti-chain algebra is provided at the end of the paper. Such modules provide an ability to share software between algorithms based on the anti-chain algebra.

Each of the three treatments of partial information representation techniques follows a similar pattern. First we provide a mathematical description of the problem to be solved using ordered structures: each of the techniques is based on an order-theoretic notion of information refinement. In particular, the algorithms all employ a common approach to optimization based on the use of operations on anti-chains. The essence of each approach is then described in terms of the anti-chain algebra and rendered in pseudo-code using the anti-chain interfaces. Once this description is given we explore correctness and optimization issues for the algorithms using the basic properties of the anti-chain algebra. For the VS and extended ATMS algorithms we conclude with generalizations of known correctness criteria and algorithms. In particular, we extend results of Hirsh [7] and Mellish [8] on the admissibility of the VS algorithm and provide a generalization and simplification of de Kleer's *choose* construct [3] for the extended ATMS.

2 Representing Sets as Anti-Chains

One way to represent a set is to maintain a list of its elements. Given an ordering for set elements, this can be optimized by maintaining the elements of a set in a structure like a balanced tree. In special circumstances a set can be maintained more indirectly as a predicate that tests set membership. This has the advantage of greater flexibility (such as the ability to represent infinite sets of elements), but it may be so general that it is impossible to implement basic operations efficiently. In this section we analyze the primitive operations for a representation that can compromise between these two approaches when the sets being represented are known to have certain order-theoretic closure properties.

To begin the discussion with an illustrative example, suppose we must maintain sets of strings of digits, supporting operations like testing whether a particular string is in a set and binary operations like taking the union or intersection of two such sets. We can order digit strings by the prefix order (for example, 01 is a prefix of 012 and 013 but not of 001) and represent them as balanced trees of strings, but it may become costly to maintain large sets in this way. We could introduce a logic capable of expressing properties of strings and then use predicates to represent sets; the efficiency of this approach will depend on the kinds of predicates we expect to use and how expensive it will be to test them. However, there are circumstances where something like a list of elements can be used (even to represent infinite sets), but where it is not essential to include all elements in order to represent the whole set. Suppose we know a special fact about the sets of strings that interest us: that they are *prefix* closed. In other words, if $s \in S$ for one of the sets S, and s is a prefix of s', then $s' \in S$ too. In this case we do not need to maintain a tree of *all* of the elements in S because the presence of some can be inferred from that of others. For instance, if $01 \in S$, then 012 and 013 are also in S. In particular, we can represent S as a set of strings S' having the property that no two strings in S' are prefixes of one another and every element of S has an element of S' as one of its prefixes. This provides a compact representation for sets which are infinite, although only sets that have finitely many distinct prefixes could be represented in this way. To determine of a string s whether it is in S, one simply checks whether it has any of the elements of S' as a prefix.

Even if checking membership is no problem, the representation will be useless if it is not possible to carry out other basic operations with it. For example, given sets S and T represented by prefixes from S' and T', how does one represent a set like $S \cup T$? This could be done by testing

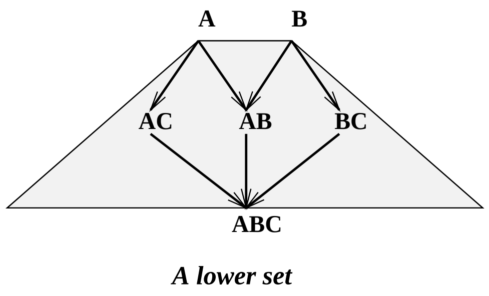

Figure 1: A Lower Subset of $\mathrm{Pwr}(P)$, where $P = \{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$. Note that $x \preceq y$ denotes $x \supseteq y$.

membership in S or T, but we can represent this test by using $S' \cup T'$. This can be optimized by removing elements $u \in S' \cup T'$ if there is an element $u' \in S' \cup T'$ such that u' is a proper prefix of u. The situation is a little more complicated for the intersection operation, but $S \cap T$ can also be calculated in terms of S', T'.

Upper sets and lower sets.

Let us now turn to identifying the idea underlying the representation employed in the above example. A *poset* is a set P together with a binary relation $\preceq$ that is reflexive ($x \preceq x$), antisymmetric ($x \preceq y$ and $y \preceq x$ implies $x = y$), and transitive ($x \preceq y$ and $y \preceq z$ implies $x \preceq z$). A set $S \subseteq P$ is said to be *downward closed* or *lower* if $x \in S$ and $y \preceq x$ implies that $y \in S$. Given a set $S \subseteq P$, there is a smallest downward-closed subset of P that contains S which is denoted by

$$\downarrow S = \{x \in P \mid x \preceq y \text{ for some } y \in S\}.$$

An example is pictured in Figure 1 It is easy to see that S is downward-closed if, and only if, $S = \downarrow S$. Dually, S is said to be *upward closed* or *upper* if

$$S = \uparrow S = \{x \in P \mid y \preceq x \text{ for some } y \in S\}.$$

It will save us some extra parentheses later if we assume that the unary operations of downward and upward closure bind more strongly than various set-theoretic operations. For example, $\downarrow S \cap \downarrow T$ is the same as $(\downarrow S) \cap (\downarrow T)$.

Notation: For a set S, the collection of all subsets of S is denoted $\mathrm{Pwr}(S)$. The collection of finite subsets of S is denoted $\mathrm{FinPwr}(S)$. $\Box$

Let us begin by assuming that the poset in question is finite and consider a specific example. Let $\mathcal{L}$ be a language of *propositional atoms* with a distinguished atom $\perp$ representing falsehood. We focus on a distinguished finite subset $\mathcal{A} \subseteq \mathcal{L}$ which we call *assumptions.* Let $\mathcal{E} = \mathrm{Pwr}(\mathcal{A})$ be the collection of subsets of $\mathcal{A}$; elements of $\mathcal{E}$ are called *environments.* Environments form a poset under the ordering $\leq$ taking $x \leq y$ if, and only if, $x \subseteq y$. Environments will arise later when we discuss the ATMS algorithms; as an intuition about their meaning, an environment is a set of assumptions whose truth would allow one to derive a given conclusion. We will be interested in representing upward-closed sets of environments and operations on these sets. For example, if $\mathcal{A} = \{A, B, C, D\}$, then the set of all environments that contain the atom A or both of the atoms C, D is

$$S = \left\{ \begin{array}{l} \{A,B,C,D\}, \{A,B,C\}, \{A,C,D\}, \{A,B,D\}, \\ \{A,B\}, \{A,C\}, \{A,D\}, \{A\}, \{C,D\}, \{B,C,D\} \end{array} \right\}.$$

We need only keep the smallest elements S' of S (the *minimal* elements), and from these we can test whether an environment x is in S by testing whether x is a superset of some $x' \in S'$. Now, the set of minimal elements of S is

$$S' = \{\{A\}, \{C, D\}\}$$

under the subset ordering (for instance, $\{A\} \preceq \{A, B\}$). A picture of this is given in Figure 2 This is what we will call a *boundary* representa-

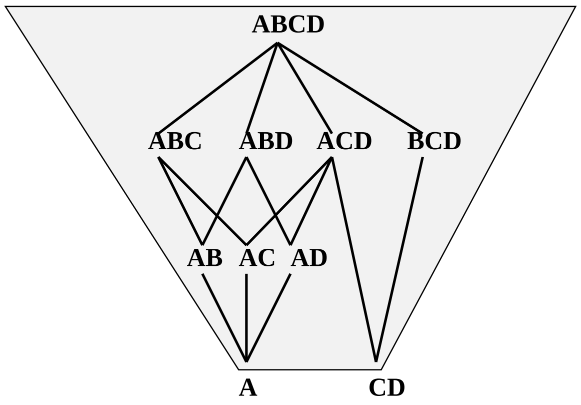

Figure 2: Representing an upper subset of $\mathrm{Pwr}(P)$ by its lower boundary, where $P = \{\mathbf{A},\mathbf{B},\mathbf{C},\mathbf{D}\}$.

tion of S because it indirectly represents S via the boundary of the set, which, in this case, is the lower boundary or set of minimal elements. In other cases we will be working with downward-closed sets, and these can be represented with their maximal elements, which form their upper boundary. And, in the case of version spaces, we will be representing a subset of a poset in terms of both an upper and a lower boundary.

Returning now to the abstract development, let us say that an element $x \in S \subseteq P$ is *maximal* in S if, for every $y \in S$, $x \preceq y$ implies $y = x$. It is said to be *minimal* in S if $y \in S$ and $y \preceq x$ implies $y = x$. Let us denote by $\max(S)$ and $\min(S)$ the respective sets of maximal and minimal elements of S. In a finite poset, lower and upper sets can be represented by their upper and lower boundaries:

Lemma 1 *Let P be a finite poset and suppose $S \subseteq P$.*

1. *if S is a lower set, then $S = \downarrow \max(S)$.*
2. *if S is an upper set, then $S = \uparrow \min(S)$.* $\square$

A generalization of this result will be needed later when we consider similar representations in an infinite poset:

Lemma 2 *Let P be a poset and suppose S' is a finite subset of P. Then*

1. $\downarrow S' = \downarrow \max(S')$.
2. $\uparrow S' = \uparrow \min(S)$. $\square$

To see that Lemma 2 *is* a generalization of Lemma 1, just note that a lower subset S of a finite poset is finite and $\downarrow S = S$. The proof of Lemma 2 is illustrative of issues that arise in the representation of infinite sets using finite boundaries.

Proof: Let us consider 1, the proof of 2 is similar. It is clear that $\downarrow \max(S') \subseteq \downarrow S'$. So take $x \in \downarrow S'$. Is there some $y \in \max(S')$ such that $x \preceq y$? Let us suppose, on the contrary, that there is no such y. Then it must be the case that x is not itself maximal in S', and therefore there is some element $x_1 \neq x$ such that $x \preceq x_1$. Assuming that we have built a chain of elements $x = x_0 \preceq x_1 \preceq \cdots \preceq x_n$ such that each $x_i \in S'$ and $x_i \neq x_j$ for distinct $i, j \leq n$, we can always extend the chain with an additional element of S' that is not in $\{x_0, \ldots, x_n\}$ because otherwise we would be forced to conclude that x_n is maximal and $x \preceq x_n$. But this implies that S' is infinite, contradicting our assumption otherwise. $\square$

Antichains.

What kinds of subsets of a poset can be the boundaries of its upper and lower subsets?

Definition: Let $P, \preceq$ be a poset. A subset $S \subseteq P$ is an *anti-chain* if it contains no comparable pair of distinct elements, that is, if $x, y \in S$ and $x \preceq y$, then $x = y$. We use the notation $\mathrm{Anti}(P)$ for the set of anti-chains over P. □

Lemma 3 *Let P be a finite poset.*

1. *The upward-closure operation $S \mapsto \uparrow S$ is a bijection (that is, one-to-one and onto mapping) between anti-chains and upward-closed subsets of P.*

2. *The downward-closure operation $S \mapsto \downarrow S$ is a bijection between anti-chains and downward-closed subsets of P.* □

The lemma can be proved by demonstrating that min and max are inverses for upward closure and downward closure respectively.

Computing basic operations on posets using antichains.

The significance of the relationship described by Lemma 3 comes from the possibility of representing operations that we would like to perform on upward-closed and downward-closed sets indirectly in terms of operations on anti-chains. Let $S - T$ be the set of elements in S that are not in T. Aside from testing set membership, here are operations that will interest us:

Difference: $\uparrow(S - T)$ where S and T are both upper sets and $\downarrow(S - T)$ where S and T are both lower sets.

Union: $S \cup T$ where S and T are both upper sets or both lower sets.

Heterogeneous intersection: $\downarrow(U \cap L)$ and $\uparrow(U \cap L)$ where U is an upper set and L is a lower set.

Homogeneous intersection: $S \cap T$ where S and T are both upper sets or both lower sets.

A few notes on the form of these operations may clarify some apparent lack of uniformity. It is easy to check that the union and intersection of a pair of upper sets is again an upper set. A similar preservation property holds for lower sets. However, $S - T$ may not be an upper set even if S and T are, so it is essential to modify the upward-set difference operation by taking the upward closure $\uparrow(S - T)$ of their ordinary set-theoretic difference. A similar consideration holds for heterogeneous intersections.

Our goal is to describe each of these mathematical operations in terms of the anti-chains by which they will be represented computationally. The description can be given mathematically so long as it is clear how the collections in question can be computed efficiently from the given description.

Let us begin with the operation $\downarrow(S-T)$ where S and T are lower sets represented by anti-chains S' and T' where $S = \downarrow S'$ and $T = \downarrow T'$. We want the anti-chain R' such that $\downarrow R' = \downarrow(S - T)$. This set R' can be shown to be the set of those elements $x \in S'$ such that there is no $y \in T'$ such that $x \preceq y$. This collection is easy to calculate: one simply takes each element of T' in turn and removes all of the elements of S' that it dominates—when all of the elements of T' have been treated in this way, we are done. Now, we want to describe this as a binary operation on anti-chains. It will be helpful to remember that this operation is intended to represent the downward closure of a difference operation but is not itself the difference of the representing anti-chains, so we need to denote it with a different symbol. We therefore write

$$\boxed{S' -^l T' = \{x \in S' \mid \forall y \in T'.\ x \not\preceq y\}}$$

where the superscript l is intended as a reminder that lower sets are being manipulated (via their representation as anti-chains). The desired property is:

$$\downarrow(S' -^l T') = \downarrow(\downarrow S' - \downarrow T').$$

Figure 3 provides a picture of the desired result. It is also possible to show that, if we define

$$\boxed{S' -^u T' = \{x \in S' \mid \forall y \in T'.\ x \not\succeq y\}}$$

where we write $x \succeq y$ to mean that $y \preceq x$, then

$$\uparrow(S' -^u T') = \uparrow(\uparrow S' - \uparrow T').$$

The union of two sets is easy to represent in these terms. If S', T' are anti-chains, then $\downarrow S' \cup \downarrow T' = \downarrow(S' \cup T')$. Unfortunately, $S' \cup T'$ may not be an anti-chain, so it is essential to take maxima:

$$\boxed{S' \cup^l T' = \max(S' \cup T')}$$

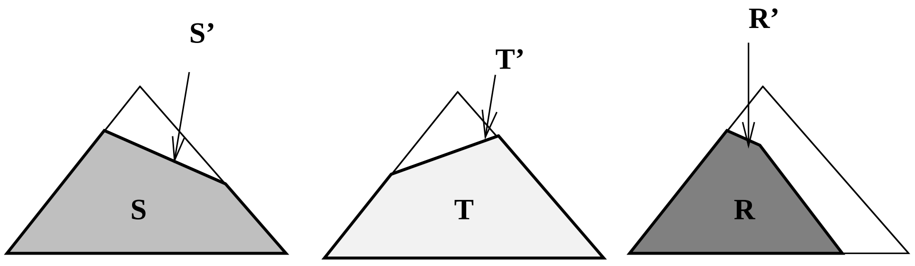

Figure 3: The lower difference $S' -^l T'$ of anti-chains S' and T' representing the lower sets S and T is the upper boundary R' of the region R in the figure.

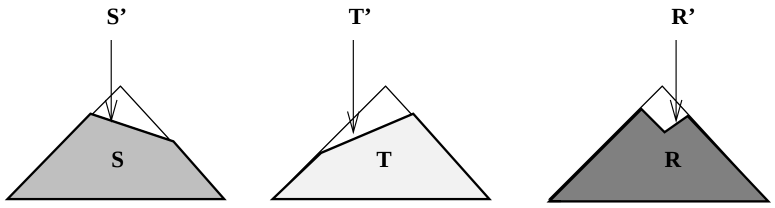

Figure 4: The lower union of anti-chains S' and T' representing the lower sets S and T is the dark region R with boundary R' shown in the figure.

Figure 4 pictures the desired result. Similarly, we define

$$\boxed{S' \cup^u T' = \min(S' \cup T')}$$

and we have

$$\downarrow\!(S' \cup^l T') = \downarrow S' \cup \downarrow T'$$
$$\uparrow\!(S' \cup^u T') = \uparrow S' \cup \uparrow T'.$$

Now, if U is an upper set and L is a lower set, then we wish to calculate (upper or lower set generated by) the intersection of U and L in terms of their boundaries. We define

$$\boxed{U' *^l L' = \{x \in L' \mid \exists y \in U'.\ y \preceq x\}}$$

and

$$\downarrow\!(U' *^l L') = \downarrow\!(\uparrow U' \cap \downarrow L').$$

We deliberately avoid using the intersection symbol $\cap$ here for heterogeneous intersection because it will be used for homogeneous intersection. The upper heterogeneous intersection has a similar representation:

$$\boxed{U' *^u L' = \{y \in U' \mid \exists x \in L'.\ y \preceq x\}}$$

which satisfies

$$\uparrow\!(U' *^u L') = \uparrow\!(\uparrow U' \cap \downarrow L'). \tag{1}$$

Figure 5 provides a picture.

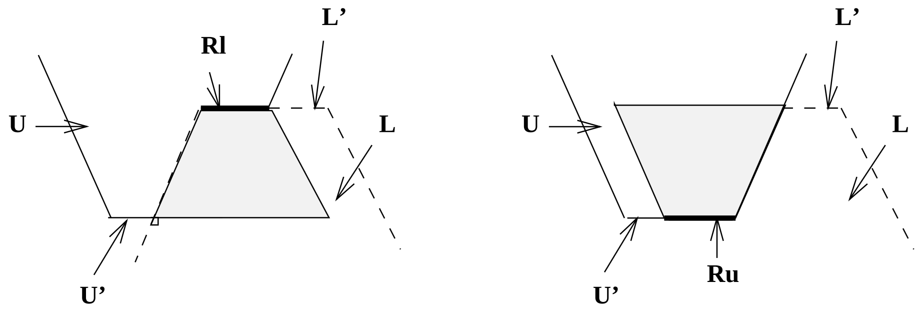

Figure 5: The lower and upper heterogeneous intersection of anti-chains U' and L' representing the upper set U and the lower set L respectively. $U' *^l L'$ is the thick upper boundary **Rl**, while $U' *^u L'$ is the thick lower boundary **Ru**.

Actually, it was not essential to include heterogeneous intersection in our collection of anti-chain operations; it can be defined in terms of the difference operation:

Proposition 4 *Let P be a poset and suppose that S' and T' are anti-chains in P, then*

1. $S' *^u T' = S' -^l (S' -^l T')$, *and*
2. $S' *^l T' = S' -^u (S' -^u T')$.

Proof: We prove the first equation; the proof of the second is similar. Let x be an element of P and let us consider what it means for it to be the case that $x \in S' -^l (S' -^l T')$. By definition, this means that $x \in S'$, but

$$\forall z \in (S' -^l T').\ x \not\preceq z. \tag{2}$$

Since S' is an anti-chain, $x \not\preceq z$ is equivalent to $x \neq z$, so the formula in Display 2 just means that $x \notin S' -^l T'$. By definition, this is the case if, and only if, there is some $y \in T'$ such that $x \preceq y$. But this, together with the fact that $x \in S'$ is just the definition of $x \in S' *^u T'$. □

The operations $\cup^l$ and $\cup^u$ can also be defined in terms of difference operations, given the usual set union and intersection operations:

Proposition 5 *Let P be a poset and suppose that S' and T' are anti-chains in P, then*

1. $S' \cup^l T' = (S' -^l T') \cup (S' \cap T') \cup (T' -^l S')$, *and*
2. $S' \cup^u T' = (S' -^u T') \cup (S' \cap T') \cup (T' -^u S')$.

Proof: We prove 1, the proof of 2 is similar.

Suppose $x \in S' \cup^l T'$. There are two cases: $x \in S'$ or $x \in T'$. Let us consider the first, the second is similar. Now, either there is an $y \in T'$ such that $x \preceq y$ or there is no such y. If there is one, then the fact that x is maximal in $S' \cup T'$ implies $x = y$. Thus $x \in S' \cap T'$. If, on the other hand, there is no such y, then $x \in S' -^l T'$ by the definition of $-^l$. Thus $\subseteq$ holds between the sets on the left and right sides of 1.

Suppose $x \in (S' -^l T') \cup (S' \cap T') \cup (T' -^l S')$. There are three possibilities. If $x \in S' -^l T'$ and there is an element $y \in S' \cup T'$ such that $x \preceq y$, then, by the definition of the lower difference operation, y cannot be an element of T'. If it is a member of S', then $x = y$ because S' is an anti-chain. Thus $x \in \max(S' \cup T')$. The second possibility is $x \in S' \cap T'$. Suppose there is some $y \in S' \cup T'$ such that $x \preceq y$. If $y \in S'$, then the fact that S' is an anti-chain means $x = y$; a similar fact holds if $y \in T'$. Thus x must be maximal in $S' \cup T'$. The third case, $x \in T' -^l S'$, has a proof similar to the first case. □

Computing homogeneous intersections on lattices.

Of the eight basic operations we set out to describe, we have now covered six: lower and upper difference ($-^l$ and $-^u$), lower and upper union ($\cup^l$ and $\cup^r$), and the heterogeneous lower and upper intersection ($*^l$ and $*^u$). This leaves the two most difficult and most interesting operations: homogeneous lower and upper intersection. Let us focus on homogeneous upper intersection; the issues with homogeneous lower intersection will be dual. Suppose we want to compute the intersection of upper sets S and T from their representations as anti-chains S' and T' where $S = \uparrow S'$ and $T = \uparrow T'$. Taking the intersection $S' \cap T'$ is clearly incorrect. To see why, consider a poset P with three elements $\{a, b, c\}$ where the only order relationships are $b \preceq a$ and $c \preceq a$. If $S = \{a, b\}$ and $T = \{a, c\}$, then $S' = \{b\}$ and $T' = \{c\}$. While $S \cap T = \{a\}$, we have $S' \cap T' = \emptyset$. In this case, the value of $S' \cap^u T'$ clearly needs to be $\{a\}$ rather than $\emptyset$. The question, therefore, is how this is calculated.

Let us consider how the intersection of upper sets should be calculated for the particular example of the poset $\mathrm{Pwr}(\mathcal{A})$ of environments under the subset ordering, where $\mathcal{A}$ is a finite set of propositional atoms. If we are given upper sets S and T of $\mathrm{Pwr}(\mathcal{A})$, then an element x in $S \cap T$ is an element of both S and T, so, if S' and T' are the minimal elements of these sets, then it is a superset of some $y \in S'$ and some $z \in T'$. This is equivalent to saying that it is a superset of $y \cup z$. Hence

$$S \cap T = \{x \mid x \supseteq y \cup z \text{ for some } y \in S' \text{ and } z \in T'\}.$$

But it is clear that the minimal elements of this collection (relative to the subset ordering) will all be sets of the form $y \cup z$ where $y \in S'$ and $z \in T'$. So the desired operation is given by

$$S' \cap^u T' = \min\{y \cup z \mid y \in S' \text{ and } z \in T'\} \tag{3}$$

To calculate the minimal elements of a finite collection R of environments is not a problem; for each element $x \in R$, compare it to each of the other elements of R removing those that are supersets and removing x itself if there is another element of R that is a subset of x.

Rather than show that the equation in Display 3 gives us the desired property, let us look at the problem more abstractly so that the equation can be applied to other posets besides $\mathrm{Pwr}(\mathcal{A})$.

Definition: A poset $P, \preceq$ is said to be a *lattice* if it satisfies the following conditions:

- There is an element $\perp$ such that $\perp \preceq x$ for each $x \in P$.

- There is an element $\top$ such that $x \preceq \top$ for each $x \in P$.
- For each pair of elements $x, y \in P$, there is an element $x \wedge y$ called the *meet* of x and y such that, $x \wedge y \preceq x$ and $x \wedge y \preceq y$ and, for any $z \in P$, if $z \preceq x$ and $z \preceq y$, then $z \preceq x \wedge y$.
- For each pair of elements $x, y \in P$, there is an element $x \vee y$ called the *join* of x and y such that, $x \preceq x \vee y$ and $y \preceq x \wedge y$ and, for any $z \in P$, if $z \succeq x$ and $z \succeq y$, then $z \succeq x \vee y$. $\Box$

This is not the place for a lengthy discussion of the properties of lattices, but it is important to note that the elements $\bot, \top$ and the operations $\wedge, \vee$ are uniquely determined by the properties ascribed to them by the definition. Other basic properties of lattices can be found in a source like [1].

Example 6 *The poset* $\mathrm{Pwr}(\mathcal{A}), \subseteq$ *of environments is a lattice where* $\bot = \emptyset$ *and* $\top = \mathcal{A}$. *The meet is* $x \wedge y = x \cap y$ *and the join is* $x \vee y = x \cup y$. $\Box$

Let us now consider how to calculate intersections of lower and upper subsets of lattices in terms of anti-chains. Let P be a lattice. For anti-chains S', T' of P, define

$$\boxed{S' \cap^l T' = \max\{x \wedge y \mid x \in S' \text{ and } y \in T'\}} \quad \text{(for lattices)}$$

and

$$\boxed{S' \cap^u T' = \min\{x \vee y \mid x \in S' \text{ and } y \in T'\}} \quad \text{(for lattices)}$$

It can be shown that the following equations are satisfied:

$$\begin{aligned} &\downarrow(S' \cap^l T') = \downarrow S' \cap \downarrow T' \\ &\uparrow(S' \cap^u T') = \uparrow S' \cap \uparrow T' \end{aligned}$$

A generalization of these facts will be proved in the next section when we consider the case in which P fails to be a lattice (Lemma 12 to be precise).

Using pairs of anti-chains to represent convex spaces.

One of the key ideas exploited in this work is the representation of another kind of subset of a poset called a *convex space*. Formally:

Definition: Let $P, \preceq$ be a poset. A subset $C \subseteq P$ is said to be a *convex space* if, for each $x, y, z \in P$, the conditions $x \preceq y \preceq z$ and $x, z \in C$ imply that $y \in C$. $\Box$

This is the order-theoretic analog of convexity in the plane, where a region C is defined to be convex if the elements on a line between any two points in C are also contained in C. Convex spaces can be described in a variety of ways.

Lemma 7 *Let P be a poset and suppose that U_1, U_2 are upper sets and L_1, L_2 are lower sets of P. Each of the following subsets of P is a convex space:*

$$U_1 \cap L_1 \qquad U_1 - L_1 \qquad L_1 - U_1 \qquad L_1 - L_2 \qquad U_1 - U_2 \quad \square$$

Figure 6 pictures four of these combinations. Two of these will concern

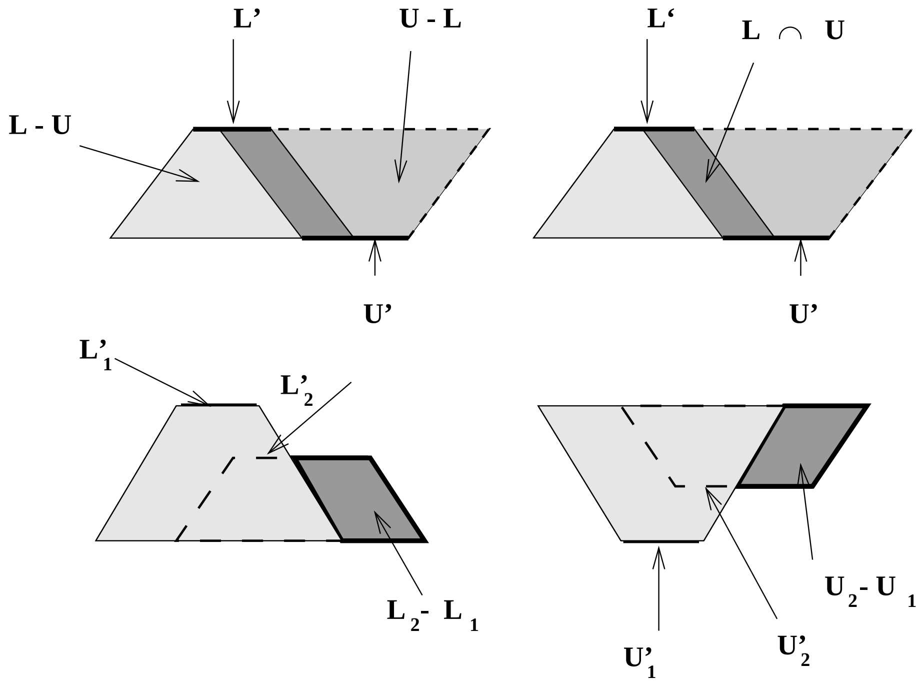

Figure 6: Representation of convex spaces using pairs of anti-chains.

us in this paper: a convex space can be represented as

- the intersection of an upper set and a lower set, or
- the difference of two upper sets.

In cases where upper and lower sets can be represented by anti-chains, it follows that convex spaces can be represented by pairs of anti-chains. For the representation of a convex space as the intersection of an upper set with a lower set, we have the following fact.

Definition: Let $P, \preceq$ be a poset and suppose $U, L \subseteq P$. Define

$$\mathcal{B}(U, L) = \{z \in P \mid x \preceq z \preceq y \text{ for some } x \in U \text{ and } z \in L.\} \quad \Box$$

Lemma 8 *Let $P, \preceq$ be a finite poset and suppose $C \subseteq P$. Then C is a convex space if, and only if,*

$$C = \mathcal{B}(\min(C), \max(C)). \Box$$

In other words, a convex space in a finite poset can be represented by its sets of maximal and minimal elements—a pair of anti-chains. The result is not true of posets in general, however. For instance, the collection of rational numbers q such that $0 < q < 1$ is a convex space, but it has no maximal or minimal element. A small generalization of the lemma would be to allow P to be any poset and restrict C to finite. However, we will also be interested in situations where C is infinite.

3 Version Spaces

Let us now consider how the anti-chain operations we have described are related to the *version space* algorithm of Mitchell [10]. The results add new insights to the ideas of Hirsh [7] and Mellish [8], which generalize Mitchell's original construction.

For our purposes a *concept space* is a set of sets P such that $\emptyset \in P$ and $\mathcal{U}P \in P$ where

$$\mathcal{U}P = \{a \mid a \in x \text{ for some } x \in P\}.$$

The elements of $\mathcal{U}P$ are called *instances* and the elements of P are called *concepts.* A concept space is partially ordered by set inclusion, that is, $x \preceq y$ iff $x \subseteq y$. It is important to note that nothing in general is known about the structure of this poset; in particular, P will not typically be the collection of all subsets of $\mathcal{U}P$ (this deviation being the 'representational bias' [9] of the concept space). If $x \preceq y$ then we say that x is *more specific* than y or we say that y is *more general* than x. A *training set*[1] over a version space P is a pair (Γ, Δ) where $\Gamma \subseteq \mathcal{U}P$ is called the set of *positive instances* and $\Delta \subseteq \mathcal{U}P$ is called the set of *negative instances.* The *version space* $\mathcal{K}(\Gamma, \Delta)$ determined by (Γ, Δ) is defined by the equation:

$$\mathcal{K}(\Gamma, \Delta) = \{x \in P \mid \Gamma \subseteq x \subseteq \overline{\Delta}\}$$

where $\overline{\Delta}$ is the complement of Δ in $\mathcal{U}P$. This collection represents the set of concepts consistent with the training set (Γ, Δ). Computationally, the goal is to calculate new version spaces as the training set is extended. In [10] this was done by an algorithm for calculating $\mathcal{K}(\Gamma \cup \{a\}, \Delta)$ and $\mathcal{K}(\Gamma, \Delta \cup \{a\})$ from $\mathcal{K}(\Gamma, \Delta)$ for any instance a. This idea was refined by Hirsh [7] to the question of how one efficiently calculates

$$\mathcal{K}(\Gamma_1 \cup \Gamma_2, \Delta_1 \cup \Delta_2)$$

in terms of $\mathcal{K}(\Gamma_1, \Delta_1)$ and $\mathcal{K}(\Gamma_2, \Delta_2)$. He aptly terms the solution for this, which can be viewed as a generalization of Mitchell's original approach, the *incremental version space merging* algorithm. Our first goal is to show how this algorithm can be understood directly in terms of the anti-chain operations defined in the previous section.

Using pairs of anti-chains to version spaces.

The key observation concerning the representation of version spaces is that the version space induced by any training set is a *convex space* and

[1] This is a slight misnomer because a training 'set' is actually a pair of sets.

may therefore be represented in one of the ways discussed earlier. In particular, [10] represents a version space as a pair of anti-chains consisting of its maximal and minimal elements. The incremental merging algorithm is defined in terms of this succinct data representation.

Lemma 9 *If P is a concept space and (Γ, Δ) is a training set, then the version space $\mathcal{K}(\Gamma, \Delta)$ is a convex space.* □

Lemmas 8 and 9 tell us that the version spaces over a finite concept space P can be represented by a pair consisting of their maximal and minimal elements. Another way to view this, in light of the correspondence between anti-chains and upper or lower sets, is to view a convex space C as a pair consisting of an upper set, $\uparrow S$, and a lower set, $\downarrow G$, where $S = \min(C)$ is the set of most specific elements of C and $G = \max(C)$ is the set of its most general elements.

The key question is how to compute the desired operations on version spaces in terms of these anti-chains. To do this, we first note that we have the following equation for training sets (Γ_1, Δ_1) and (Γ_2, Δ_2):

$$\mathcal{K}(\Gamma_1 \cup \Gamma_2, \Delta_1 \cup \Delta_2) = \mathcal{K}(\Gamma_1, \Delta_1) \cap \mathcal{K}(\Gamma_2, \Delta_2). \tag{4}$$

This means that it suffices to be able to compute the intersection of version spaces *in terms of the pairs of anti-chains that represent them.* We can describe how to do this quite succinctly in terms of our collection of anti-chain operations by the following definition:

$$\boxed{\begin{aligned}&(S_1, G_1) \cap^c (S_2, G_2)\\ &\quad = ((S_1 \cap^u S_2) *^u (G_1 \cap^l G_2),\ (S_1 \cap^u S_2) *^l (G_1 \cap^l G_2))\end{aligned}} \tag{5}$$

Correctness of the equation is described by the following:

Theorem 10 *Let P be a finite concept space that is a lattice and suppose (Γ_1, Δ_1) and (Γ_2, Δ_2) are training sets with*

$$\begin{array}{ll} S_1 = \min(\mathcal{K}(\Gamma_1, \Delta_1)) & G_1 = \max(\mathcal{K}(\Gamma_1, \Delta_1)) \\ S_2 = \min(\mathcal{K}(\Gamma_2, \Delta_2)) & G_2 = \max(\mathcal{K}(\Gamma_2, \Delta_2)). \end{array}$$

If $(S_3, G_3) = (S_1, G_1) \cap^c (S_2, G_2)$ and $C = \mathcal{K}(\Gamma_1 \cup \Gamma_2, \Delta_1 \cup \Delta_2)$, then $\min(C) = S_3$ and $\max(C) = G_3$. □

The theorem is an immediate consequence of following:

Lemma 11 *Let C_1, C_2 be convex spaces that are subsets of a finite lattice P and*

$$(S', T') = (\min(C_1), \max(C_1)) \cap^c (\min(C_2), \max(C_2)).$$

Table 1: Incremental Version Space Merging Algorithm.

```
function mergeVS((S1, G1), (S2, G2)) =
  let value U = S1 $\cap^u$ S2
      and   L = G1 $\cap^l$ G2
      and   S3 = U $*^u$ L
      and   G3 = S3 $*^l$ L
  in  (S3, G3)
  endlet
```

Then $\min(C_1 \cap C_2) = S'$ *and* $\max(C_1 \cap C_2) = T'$. *In particular,*

$$C_1 \cap C_2 = \mathcal{B}(S', T'). \quad \Box$$

We omit the proof since it is similar to the one we give for a more general result (Lemma 13).

From a computational perspective, there is some redundancy in Equation (5) since the anti-chains $S = S_1 \cap^u S_2$ and $G = G_1 \cap^l G_2$ are apparently calculated twice each. Moreover, there is an optimization one can make in calculating the second component if one is given the value of the first. A more realistic algorithmic presentation of version space merging is given in Table 1 where the program is described in pseudo-code using the appropriate four anti-chain operations. The basic constructs in this pseudo-code will be used in later examples as well. As a brief explanation, pairs (and tuples) are written with parentheses and commas: `(S, G)`. The form

```
function f(x) = $E$
```

declares a function `f` with formal parameter `x` and body E. It will often be useful to describe the formal parameter as a pattern. For example, the function `mergeVS` takes a pair of pairs as an argument. The form

```
let $D$ in $E$ endlet
```

evaluates expression E after establishing the bindings from declaration D. The declaration form

```
value $x_1$ = $E_1$
and   $x_2$ = $E_2$
      ...
and   $x_n$ = $E_n$
```

binds each x_i in the environment obtained after establishing bindings for x_j where $j < i$.

The calculation in Table 1 is essentially the same as those in [10, 7]. It differs from the value described in Equation (5) in calculating $S_1 \cap^u S_2$ and $G_1 \cap^l G_2$ only once, of course, but also by using S_3 to compute G_3 rather than using $U = S_1 \cap^u S_2$ for this purpose. To see that it is equivalent to the value described by Equation (5), we need only establish the following equation

$$(U *^u L) *^l L = U *^l L. \tag{6}$$

for anti-chains U and L. To see that this implies that the value `(S3, G3)` computed in Table 1 matches the value calculated in Equation (5), just let $U = S_1 \cap^u S_2$ and $L = G_1 \cap^l G_2$ and substitute using Equation (6). To see why (6) holds, let us analyze what it means to have $x \in (U *^u L) *^l L$. Unfolding the definitions of upper and lower heterogeneous intersection yields the assertion:

$$x \in L \text{ and } \exists y \in (U *^u L).\ y \preceq x$$

which means

$$x \in L \text{ and } \exists y \in U.\ (\exists x_0 \in L.\ y \preceq x_0) \text{ and } y \preceq x.$$

But this is clearly equivalent to

$$x \in L \text{ and } \exists y \in U.\ y \preceq x$$

which is the same as $x \in U *^l L$.

The version space learning algorithm itself proceeds by repeated merging of version spaces. The algorithm assumes that we are given a way to obtain new information in the form of a version space; this may be done by extending a training set by examining a new instance, but any method—including the exploitation of domain knowledge—would suit the algorithm. The new information is merged with the old until the desired level of accuracy is achieved. Pseudo-code for this process is given in Table 2. The code there uses a form

```
if $B$ then $E_1$ else $E_2$ endif
```

which evaluates a boolean B and, depending on whether its value is true or false, evaluates expression E_1 or E_2 respectively, and returns the resulting value. The `learnVS` function is used by being invoked on an initial version space.

Table 2: Version Space Learning.

```
function learnVS(G,S) =
  if (G,S) is good enough
  then (G,S)
  else let value (newG, newS) be learned from new information
           and   (betterG, betterS)
                    = mergeVS((G,S), (newG, newS))
       in  learnVS(betterG, betterS)
       endlet
  endif
```

A simple concept learning example.

We illustrate the theoretical concepts introduced in this section in the context of a simple concept learning problem adapted from [10]. The instance space $\mathcal{UP}$ is a set of objects identified by their shape and color. The concept space P is a set of subsets of $\mathcal{UP}$.

$$\mathcal{UP} = \{\text{Red}\square, \text{Red}\diamond, \text{Blue}\square, \text{Blue}\diamond\}$$

$$P = \left\{ \begin{array}{l} \emptyset, \{\text{Red}\square\}, \{\text{Red}\diamond\}, \{\text{Blue}\square\}, \{\text{Blue}\diamond\}, \\ \{\text{Red}\square, \text{Red}\diamond\}, \{\text{Red}\square, \text{Blue}\square\}, \\ \{\text{Blue}\square, \text{Blue}\diamond\}, \{\text{Red}\diamond, \text{Blue}\diamond\}, \mathcal{UP} \end{array} \right\}$$

Note that only 10 of the 16 possible subsets are represented in P. This concept space bias provides the means of making non-trivial generalizations of observed instances. The elements of P are ordered by subset inclusion. Thus, $\{\text{Red}\square\} \preceq \{\text{Red}\square, \text{Blue}\square\}$.

Consider the version space generated by the training set $(\emptyset, \{\text{Blue}\diamond\})$:

$$\begin{array}{rcl} \mathcal{K}(\emptyset, \{\text{Blue}\diamond\}) & = & \{x \in P \mid \emptyset \subseteq x \subseteq \mathcal{UP} - \{\text{Blue}\diamond\}\} \\ & = & \left\{ \begin{array}{l} \emptyset, \{\text{Red}\square\}, \{\text{Red}\diamond\}, \\ \{\text{Blue}\square\}, \{\text{Red}\square, \text{Red}\diamond\}, \\ \{\text{Red}\square, \text{Blue}\square\} \end{array} \right\} \end{array}$$

The anti-chains that represent the minimal and maximal boundaries of this version space are:

$$S_1 = \{\emptyset\}$$
$$G_1 = \{\{\text{Red}\square, \text{Red}\diamond\}, \{\text{Red}\square, \text{Blue}\square\}\}$$

Now consider the version space generated by $(\{\text{Red}\Box\}, \emptyset)$:

$$\begin{aligned}
\mathcal{K}(\{\text{Red}\Box\}, \emptyset) &= \{x \in P \mid \{\text{Red}\Box\} \subseteq x \subseteq \mathcal{UP}\} \\
&= \left\{ \begin{array}{l} \{\text{Red}\Box\}, \{\text{Red}\Box, \text{Red}\Diamond\}, \\ \{\text{Red}\Box, \text{Blue}\Box\}, \\ \{\text{Red}\Box, \text{Blue}\Box, \text{Red}\Diamond, \text{Blue}\Diamond\} \end{array} \right\}
\end{aligned}$$

The anti-chains that represent the boundaries of this version space are:

$$S_2 = \{\{\text{Red}\Box\}\}$$
$$G_2 = \{\{\text{Red}\Box, \text{Red}\Diamond, \text{Blue}\Box, \text{Blue}\Diamond\}\}$$

We now show the steps in the computation of $\mathcal{K}(\{\text{Red}\Box\}, \{\text{Blue}\Diamond\}) = \mathcal{K}(\emptyset, \{\text{Blue}\Diamond\}) \cap \mathcal{K}(\{\text{Red}\Box\}, \emptyset)$ in terms of the anti-chains that represent them (Table 1).

$$\mathcal{K}(\{\text{Red}\Box\}, \{\text{Blue}\Diamond\}) = \{\{\text{Red}\Box\}, \{\text{Red}\Box, \text{Red}\Diamond\}, \{\text{Red}\Box, \text{Blue}\Box\}\}$$

$$\begin{aligned}
S_1 \cap^u S_2 &= \{\{\text{Red}\Box\}\} \\
G_1 \cap^l G_2 &= \{\{\text{Red}\Box, \text{Red}\Diamond\}\}, \{\text{Red}\Box, \text{Blue}\Box\}\} \\
S_3 &= (S_1 \cap^u S_2) *^u (G_1 \cap^l G_2) \\
&= \{y \in (S_1 \cap^u S_2) \mid \exists x \in (G_1 \cap^l G_2)\ y \preceq x\} \\
&= \{\{\text{Red}\Box\}\} \\
G_3 &= (S_3 *^l (G_1 \cap^l G_2)) \\
&= \{x \in (G_1 \cap^l G_2) \mid \exists y \in S_3 .\ y \preceq x\} \\
&= \{\{\text{Red}\Box, \text{Red}\Diamond\}, \{\text{Red}\Box, \text{Blue}\Box\}\}
\end{aligned}$$

We can easily see that $\mathcal{B}(S_3, G_3) = \mathcal{K}(\{\text{Red}\Box\}, \{\text{Blue}\Diamond\})$.

More general concept spaces.

We have now shown how operations on version spaces are calculated if the concept space is a finite lattice, but is there a way to work with spaces that are not finite lattices? We now describe a necessary and sufficient condition for when we are assured that the incremental update algorithm is usable, at least in principle. The key idea is to identify what we need to be able to compute; this is given by the following:

Definition: Let $P, \preceq$ be a poset $x, y \in P$. The *quasi-meet* of x and y is defined by the equation

$$x \tilde{\wedge} y = \max\{z \in P \mid z \preceq x \text{ and } z \preceq y\}$$

and the *quasi-join* of x and y is defined by

$$x \tilde{\vee} y = \min\{z \in P \mid z \succeq x \text{ and } z \succeq y\} \quad \Box$$

In [8], $\downarrow x \tilde{\wedge} y$ is called $bb(x, y)$ and $\uparrow x \tilde{\vee} y$ is called $aa(x, y)$.

If P is a lattice, then it is easy to check that $x \tilde{\wedge} y = \{x \wedge y\}$ and $x \tilde{\vee} y = \{x \vee y\}$ so, in the event that we are dealing with lattices, quasi-meets and quasi-joins basically correspond to meets and joins. However, it is not always the case that quasi-meets and quasi-joins are singletons. A graphical representation using a Hasse diagram is given in Figure 7 where $x \tilde{\vee} y = \{g_1, \ldots, g_m\}$ and $x \tilde{\wedge} y = \{s_1, \ldots, s_n\}$. When the poset

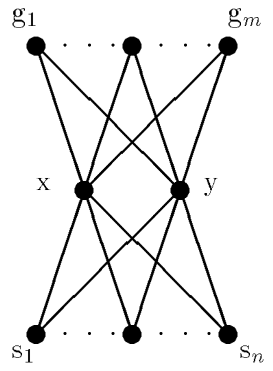

Figure 7: Quasi-meets and quasi-joins.

P is a concept space, the elements of set $x \tilde{\vee} y$ are sometimes called the 'most specialized generalizations' of x, y and those of $x \tilde{\wedge} y$ are called 'most general specializations' of x, y.

It is not sufficient simply to know how to compute quasi-meets and quasi-joins; it is essential to know that these operations can be used to compute anti-chains that are needed to represent version spaces. However, this is a special property of the concept space. Just as we defined a lattice to be a poset on which there are operations $\wedge, \vee$ with special properties, we need an analogous

Definition: A poset P is said to have *property W* if

- $\min(P)$ is finite and $P = \uparrow \min(P)$ and,
- for each $x, y \in P$, the quasi-meet $x \tilde{\wedge} y$ is finite and

$$\downarrow(x \tilde{\wedge} y) = \{z \in P \mid z \preceq x \text{ and } z \preceq y\}$$

It is said to have *property M* if

- $\max(P)$ is finite and $P = \downarrow \max(P)$, and

- for each $x, y \in P$, the quasi-join $x \tilde{\vee} y$ is finite and

$$\uparrow(x \tilde{\vee} y) = \{z \in P \mid z \succeq x \text{ and } z \succeq y\}$$

A poset that has both properties is said to have *property MW.* □

Property M is familiar from ideas in topology (where it is an order-theoretic formulation of an important property of compact subsets called 'coherence' [15]) and in domain theory (where it is a necessary condition for the bases of domains with good closure properties [14]). Mellish identified these conditions in [8]; he too noted the need for finite quasi-meets and quasi-joins of pairs of elements of P, and by fiat introduced a top and bottom element for P so that the first part of the M and W properties hold.

Given an MW poset P, we can now express in greater generality how the intersection of two upper sets or two lower sets can be carried out in terms of their representation as anti-chains. We define

$$\boxed{S' \cap^l T' = \max(\bigcup\{x \tilde{\wedge} y \mid x \in S' \text{ and } y \in T'\})} \quad \text{(for MW posets)}$$

and

$$\boxed{S' \cap^u T' = \min(\bigcup\{x \tilde{\vee} y \mid x \in S' \text{ and } y \in T'\})} \quad \text{(for MW posets)}$$

Note, in particular, that although P may be infinite, if S' and T' are finite, then $S' \cap^l T'$ and $S' \cap^u T'$ are also finite. The definition of $\cap^c$ is the same as given in Equation 5 although the operations used there should be taken on MW posets. That is, the definitions of $*^l$ and $*^u$ are unchanged, but those for $\cap^l$ and $\cap^u$ are the ones just given.

Lemma 12 *Let P be a poset and suppose S', T' are anti-chains in P.*

1. *If P has property W, then $\downarrow(S' \cap^l T') = \downarrow S' \cap \downarrow T'$.*
2. *If P has property M, then $\uparrow(S' \cap^u T') = \uparrow S' \cap \uparrow T'$.*

Proof: We prove (1); the proof of (2) is dual. First suppose that $x \in \downarrow(S' \cap^l T')$. Then there are elements y, u, v such that $x \preceq y$ where $y \in u \tilde{\wedge} v$ and $u \in S'$ and $v \in T'$. This means that $x \preceq u$ and $x \preceq v$ so $x \in \downarrow S' \cap \downarrow T'$.

Now suppose that $x \in \downarrow S'$ and $x \in \downarrow T'$. Then there are elements $u_0 \in S'$ and $v_0 \in T'$ such that $x \preceq u_0, v_0$. By property W we must therefore have $x \in \downarrow(u_0 \tilde{\wedge} v_0)$, so x is an element of $\downarrow X$ where

$$X = \bigcup\{u \tilde{\wedge} v \mid x \in S' \text{ and } y \in T'\}$$

The set X is finite, so, by Lemma 2, $\downarrow X = \downarrow(\max(X)) = \downarrow(S' \cap^l T')$ as desired. □

Lemma 13 *Suppose P is an MW poset and $C_1, C_2 \subseteq P$ are convex spaces where the sets*

$$\begin{array}{ll} S_1 = \min(C_1) & G_1 = \max(C_1) \\ S_2 = \min(C_2) & G_2 = \max(C_2). \end{array}$$

are finite and the equations $C_1 = \mathcal{B}(S_1, G_1)$ and $C_2 = \mathcal{B}(S_2, G_2)$ hold. If

$$(S', T') = (\min(C_1), \max(C_1)) \cap^c (\min(C_2), \max(C_2)),$$

then $\min(C_1 \cap C_2) = S'$ and $\max(C_1 \cap C_2) = T'$. In particular,

$$C_1 \cap C_2 = \mathcal{B}(S', T').$$

Proof: The desired result follows from Lemma 8 if we can show that

$$\begin{array}{l} \min(C_1 \cap C_2) = (S_1 \cap^u S_2) *^u (G_1 \cap^l G_2) \text{ and} \\ \max(C_1 \cap C_2) = (S_1 \cap^u S_2) *^l (G_1 \cap^l G_2) \end{array}$$

Let us do the first of these, the second has a similar proof. Starting with Equation 1 and Lemma 13 we calculate:

$$\begin{array}{ll} \multicolumn{2}{l}{\uparrow((S_1 \cap^u S_2) *^u (G_1 \cap^l G_2))} \\ = & \uparrow((\uparrow S_1 \cap \uparrow S_2) \cap (\downarrow G_1 \cap \downarrow G_2)) \\ = & \uparrow((\uparrow S_1 \cap \downarrow G_1) \cap (\uparrow S_2 \cap \downarrow G_2)) \\ = & \uparrow(\mathcal{B}(S_1, G_1) \cap \mathcal{B}(S_2, G_2)) \\ = & \uparrow(C_1 \cap C_2) \end{array}$$

It is not difficult to check that if U' is an anti-chain and $U = \uparrow U'$, then $U' = \min(U)$. The desired equation therefore follows. □

Necessity of properties M and W.

The MW property is not a difficult condition to satisfy. For instance, any finite poset has property MW and any lattice has property MW. But are there examples of concept spaces on which incremental version space merging could be used but where MW is not satisfied? We now show that it can essentially be claimed that concept spaces for which the anti-chain representation on which the algorithm in Table 1 depends can only make sense for a poset that satisfies the MW property.

The key assumption underlying the anti-chain representation of a convex space is that the convex spaces that are represented in this way have the form $\mathcal{B}(S, G)$ where S and G are *finite* anti-chains. Some vocabulary is helpful here:

Definition: Let P be a poset. A lower subset S of P is said to be *finitely representable* if $\max(S)$ is finite and $S = \downarrow\max(S)$. A upper subset $T \subseteq P$ is said to be finitely representable if $\min(T)$ is finite and $T = \uparrow\min(T)$. A convex space $C \subseteq P$ is said to be finitely representable if $\max(C)$ and $\min(C)$ are both finite and $C = \mathcal{B}(\min(C), \max(C))$. □

In these terms it is possible to express succinctly the Admissibility Theorem for version spaces:

Theorem 14 *(Admissibility) Let P be a concept space that satisfies property MW. If, for every instance a, the convex spaces $\mathcal{K}(\{a\}, \emptyset)$ and $\mathcal{K}(\emptyset, \{a\})$ are finitely representable, then so is $\mathcal{K}(\Gamma, \Delta)$ for any training set (Γ, Δ).*

Proof: Note first that $\mathcal{K}(\emptyset, \emptyset) = \mathcal{B}(\min(P), \max(P))$ is finitely representable by the conditions for properties M and W concerning the collections $\min(P)$ and $\max(P)$. Equation (4) says that it is possible to express other collections as intersections of convex spaces of the forms $\mathcal{K}(\{a\}, \emptyset)$ and $\mathcal{K}(\emptyset, \{a\})$. Based on Lemma 13 and our assumption that such collections are finitely representable, it follows that $\mathcal{K}(\Gamma, \Delta)$ must also be finitely representable. □

Theorem 15 *Let P be a poset.*

1. *If the set $\min(P)$ is finite with $P = \uparrow\min(P)$ and intersections of finitely-representable upwards-closed subsets are finitely representable, then P has property W.*

2. *If the set $\max(P)$ is finite with $P = \downarrow\max(P)$ and intersections of finitely-representable upwards-closed subsets are finitely representable, then P has property M.*

3. *If the poset P is itself a finitely-representable convex space and if intersections of its finitely-representable convex subsets are finitely representable, then P has properties M and W.*

Proof: We prove (1); the others have similar proofs. Let $x, y \in P$ and let $L = \downarrow\{x\} \cap \downarrow\{y\}$. Since this is an intersection of finitely representable sets it must be finitely representable, so suppose L' is an anti-chain such that $L = \downarrow L'$. In this case $L' = \max(L)$. Since $L = \{z \mid z \preceq x \text{ and } z \preceq y\}$, it follows that $L' = x \tilde{\wedge} y$. Hence

$$\downarrow(x \tilde{\wedge} y) = \{z \in P \mid z \preceq x \text{ and } z \preceq y\}$$

and this means that P has property W. □

Theorem 14 and Theorem 15 significantly extend the known theory of admissibility presented in [10, 7, 8]. They provide a direct means for verifying whether the antichain representation adopted by the version space algorithm is correct for an arbitrary training sequence and a concept language that satisfies the MW property in terms of its behavior on each element of the set of training instances.

A complex concept learning example.

We now turn to a more complex example of a concept space in order to illustrate the role that properties M and W may play in the admissibility of the version space algorithm. Theorem 15 says that if intersections of finitely representable (fr) subsets of an fr poset P are always fr, then P has property MW. Since the version space algorithm relies on intersecting fr subsets to get fr subsets, this result is clearly relevant. However, it may be the case that the algorithm does not actually need to take intersections of *all* possible pairs of fr subsets. If the intersections it does take are always finite, then no problem will arise. To illustrate how this can happen, we examine the task of learning axis-parallel rectangles on a plane from labeled points on the plane. This problem was introduced in [10] and has applications in the geometric analysis of subsymbolic learning methods.

Consider an integer grid imposed on the xy plane. Labeled examples are points (x, y) on the grid marked as either being inside or outside the target axis-parallel rectangle. The target rectangle is represented as a conjunction of two closed intervals over the natural numbers $\mathcal{N}$. We now formally define the set $\mathcal{UP}$ of instances and the set P of concepts composed of elements that are subsets of $\mathcal{UP}$. As in the previous concept learning example, representational bias is introduced in P to permit non-trivial generalization of instances. In particular, P consists of those subsets of $\mathcal{UP}$ that can be represented as the product of a pair of closed, half-open, or open intervals on x and y. To be precise, suppose $R \in \{<, \leq\}$, and define:

$$\begin{aligned} P &= \{\{(x, y) \mid [l\ R\ x\ R\ r] \wedge [b\ R\ y\ R\ t]\} \mid \ \ l, r, b, t \in \mathcal{N} \cup \{\infty\}\} \\ \mathcal{UP} &= \{(x, y) \mid x \in \mathcal{N} \text{ and } y \in \mathcal{N}\} \end{aligned}$$

The elements of P are (points in) open, half-open, or closed rectangles corresponding to whether both defining intervals are open, at least one interval is half-closed, or both intervals are closed. This poset has the empty rectangle as a least element and $\mathcal{UP}$ as a greatest one. Let us define the $\wedge$ operation between two elements p and q in P to yield the largest axis-parallel rectangle on the grid that is contained in both p and q. Let us define the operation $\vee$ between elements p and q in P to be the smallest rectangle on the grid that contains them both *if there is one.*

To see where this proviso about existence comes from, consider the rectangles

$$R_1 = \{(x,y) \mid 0 < x < 20 \text{ and } 0 < y < 30\}$$
$$R_2 = \{(x,y) \mid 10 \leq x \leq 30 \text{ and } 0 \leq y \leq 30\}.$$

Their union is not an element of P, and it is not difficult to see that there is *no least* element of P that contains them both. That is, $R_1 \vee R_2$ is undefined. But more that this is true: property M fails! To see why, note that it is not just the case that there is no *least* element in P containing $R_1 \cup R_2$; there is also no *minimal* element in P containing it.

Let us put these concerns aside for now and focus on how the algorithms in Tables 1 and 2 are carried out using the partially defined operations for conjunction and disjunction. The target concept is a closed rectangle defined as $l \leq x \leq r$ and $b \leq y \leq t$. Should $l > r$ or $b > t$ we have the empty rectangle. The ordering $\preceq$ on P is that of set inclusion $\subseteq$. Because of the special form of the elements in P, we can check set inclusion by checking for inclusion in the x and y intervals. We will use this fact in the construction of anti-chain representations of the version space.

We calculate the version space given by the training set having the points $(12,12), (11,11)$ as a positive instances and the point $(13,13)$ as a negative one. Let us assume that they are learned in the following order: $+(12,12)$, $-(13,13)$, $+(11,11)$. By definition,

$$\mathcal{K}(\{(12,12)\}, \emptyset) \quad = \quad \{S \in P \mid \{(12,12)\} \subseteq S \subseteq \mathcal{UP}\}$$

The anti-chains S_1 and G_1 representing $K(\{(12,12)\}, \emptyset)$ are sets of elements of P. In this example, S_1 and G_1 are singletons: S_1 contains a point rectangle at $(12,12)$ and G_1 is the half-open rectangle with its bottom left corner at $(0,0)$ and its top right corner at (∞, ∞).

$$\begin{aligned} S_1 &= \{(x,y) \mid 12 \leq x \leq 12 \text{ and } 12 \leq y \leq 12\} \\ G_1 &= \{(x,y) \mid 0 \leq x < \infty \text{ and } 0 \leq y < \infty\} \end{aligned}$$

Now, $(13,13)$ is a point outside of our target closed rectangle so, by definition:

$$\mathcal{K}(\emptyset, \{(13,13)\}) = \{S \in P \mid \emptyset \subseteq S \subseteq \mathcal{UP} - \{(13,13)\}\}$$

While S_2 is a singleton consisting of the empty rectangle, G_2 consists of four elements in P denoting four rectangles on the plane that exclude the point $(13, 13)$:

$$\begin{aligned} S_2 &= \{\emptyset\} \\ G_2 &= \{\{(x,y) \mid 0 \leq x < 13 \text{ and } 0 \leq y < 13\}, \\ &\quad \{(x,y) \mid 0 < x < 13 \text{ and } 13 < y < \infty\}, \\ &\quad \{(x,y) \mid 13 < x < \infty \text{ and } 0 < y < 13\}, \\ &\quad \{(x,y) \mid 13 < x < \infty \text{ and } 13 < y < \infty\}\} \end{aligned}$$

We now calculate the anti-chains representing the version space

$$\mathcal{K}(\{(12,12)\}, \{(13,13)\}) = \mathcal{K}(\{(12,12)\}, \emptyset) \cap \mathcal{K}(\emptyset, \{(13,13)\})$$

using the construction in Table 1. Each element in an anti-chain is an open, half-open, or closed rectangle. The pairwise $\vee$ between the elements in S_1 and S_2 yields the point rectangle in S_1. The pairwise $\wedge$ operation between elements in G_2 and the element in G_1 yields the elements in G_2 in turn.

$$\begin{aligned} S_1 \cap^u S_2 &= \min\{p \vee q \mid p \in S_1 \text{ and } q \in S_2\} \\ &= \{(x,y) \mid 12 \leq x \leq 12 \text{ and } 12 \leq y \leq 12\} \\ G_1 \cap^l G_2 &= \max\{p \wedge q \mid p \in S_1 \text{ and } q \in S_2\} \\ &= \{\{(x,y) \mid 0 \leq x < 13 \text{ and } 0 \leq y < 13\}, \\ &\quad \{(x,y) \mid 0 < x < 13 \text{ and } 13 < y < \infty\}, \\ &\quad \{(x,y) \mid 13 < x < \infty \text{ and } 0 < y < 13\}, \\ &\quad \{(x,y) \mid 13 < x < \infty \text{ and } 13 < y < \infty\}\} \end{aligned}$$

To obtain the anti-chains S_3 and G_3 representing the boundaries of

$$\mathcal{K}(\{(12,12)\}, \{(13,13)\})$$

we complete the last two steps in the algorithm in Table 1:

$$\begin{aligned} S_3 &= (S_1 \cap^u S_2) *^u (G_1 \cap^l G_2) \\ &= \{p \in (S_1 \cap^u S_2) \mid \exists q \in (G_1 \cap^l G_2)\ p \preceq q\} \\ &= \{(x,y) \mid 12 \leq x \leq 12 \text{ and } 12 \leq y \leq 12\} \\ G_3 &= (S_1 \cap^u S_2) *^l (G_1 \cap^l G_2) \\ &= \{q \in (G_1 \cap^l G_2) \mid \exists p \in (S_1 \cap^u S_2)\ p \preceq q\} \\ &= \{(x,y) \mid 0 \leq x < 13 \text{ and } 0 \leq y < 13\} \end{aligned}$$

As expected, S_3 is the point rectangle at $(12, 12)$ and G_2 is the half-open rectangle with its bottom left corner at $(0, 0)$ and its top right corner at $(13, 13)$ where $(13, 13)$ is not included in this rectangle.

The version space for the positive instance $(11, 11)$ is represented by the pair (S_4, G_4) where

$$\begin{aligned} S_4 &= \{(x, y) \mid 11 \leq x \leq 11 \text{ and } 11 \leq y \leq 11\} \\ G_4 &= \{(x, y) \mid 0 \leq x < \infty \text{ and } 0 \leq y < \infty\} \end{aligned}$$

When we merge this with (S_3, G_3), the upper boundary remains G_3, but the new lower boundary is

$$S_5 = \{(x, y) \mid 11 \leq x \leq 12 \text{ and } 11 \leq y \leq 12\},$$

since this is the value of $S_3 \cap^u S_4$, a closed square with corners at $(11, 11)$ and $(12, 12)$.

Let us return now to the question of why the algorithm seems to work (at least on the example) even though property M fails. The need for property M arises from the use in the algorithm of quasi-joins to calculate lower set boundaries, which serve as the lower boundaries of the convex version spaces. However, it is not hard to see that not every concept in P could arise as an element of a lower boundary S of a version space when only training sequences of points are used to generate version spaces. To see why, note that the lower boundary of a version space will always be a *single* closed rectangle, which will, in fact, be the smallest (most specific) rectangle containing the positive instances of the training sequence. Our counter-example to property M was obtained by attempting to take the union of a closed rectangle with an open one—a situation that will not arise in the algorithm unless new convex spaces are learned by some other means than training sequences of points. By contrast, the upper boundaries of version spaces in this concept space are always open rectangles (that is, rectangles that do not include their finite boundaries). This heterogeneity has the consequence that we cannot stipulate that a concept is 'learned' when S and G in the version space *coincide* since this will never happen! So, referring to Table 2, the test `good_enough` must be defined to test that S and G are singleton sets such that the difference between them is a border of width 1 around S.

In general terms, the success of the version space representation for this concept space can be seen as relying on the fact that upper and lower boundaries lie in special subsets of the concept space. Another way to say this, is that the specific concepts are represented in a concept space of closed rectangles while the general concepts are represented in one of open rectangles. To incorporate this into a general theory such as the one we have described in this section would require generalizing our view of version spaces to accommodate *three* partial orders: one for upper boundaries, one for lower boundaries, and one for comparing upper and lower boundaries. The first of these must satisfy property M and the second must satisfy property W. The order for comparing upper and lower does not need to satisfy any order-theoretic properties, but, of course, we must be able to compute it in order to execute the version space merging algorithm, where it will be used to compute heterogeneous intersections. We will not attempt a further elaboration of this generalization in this paper.

4 Assumption-Based Truth Maintenance Systems

An Assumption-Based Truth Maintenance System (ATMS) is a structure introduced by Johann de Kleer [2, 3] which is intended to compute sets of assumptions on which a conclusion can be based relative to a given theory. Our goal in this section is to show how the computations involved in the ATMS can be expressed in terms of anti-chains. This provides a semantic basis for understanding the ATMS and reveals efficient representations for the calculations.

To describe the ATMS, we need some ideas from logic. We work with a language $\mathcal{L}$ of propositional atoms that includes an atom $\perp$ standardly interpreted as falsehood. In general, atoms will be denoted with lower case letters a, b, c from the beginning of the Latin alphabet. *Propositions* (or *formulas*) ϕ, ψ are built from atoms using the usual logical connectives $\Rightarrow$ for implication, $\neg$ for negation, $\wedge$ for conjunction, and $\vee$ for disjunction. A *theory* $\mathcal{F}$ is a set of propositions. A *model* M is a subset of $\mathcal{L} - \{\perp\}$ where an atom a is interpreted as being true if, and only if, $a \in M$. We write $M \models \phi$ if M interprets ϕ as true with the usual truth table interpretation of the logical connectives. We write $\mathcal{F} \models \phi$ if each model of the elements of $\mathcal{F}$ is also a model of ϕ. We say that a theory $\mathcal{F}$ is *inconsistent* if it has no models or, equivalently, if $\mathcal{F} \models \perp$.

As we discussed in an example earlier, we let $\mathcal{A}$ be a finite subset of $\mathcal{L}$ whose elements we call *assumptions* and define the poset of *environments* $\mathcal{E}$ to be $\mathrm{Pwr}(\mathcal{A})$ under the subset ordering. Assumption atoms may be written using upper case letters A, B, C to help distinguish them from general atoms. Environments will be denoted with lower case letters x, y, z from near the end of the alphabet.

Given a theory $\mathcal{F}$ and assumption set $\mathcal{A}$, an ATMS is designed to efficiently calculate the collection of *nogoods*, defined as

$$N(\mathcal{F}) = \{x \in \mathcal{E} \mid \mathcal{F} \cup x \models \perp\}$$

and, for each $a \in \mathcal{L}$, the set

$$V(\mathcal{F}, a) = \{x \in \mathcal{E} \mid \mathcal{F} \cup x \not\models \perp \text{ and } \mathcal{F} \cup x \models a\}$$

In practice, this function is invoked more often on different a's than on different $\mathcal{F}$'s so the primary computational goal of the ATMS is to calculate the function

$$V_{\mathcal{F}}(a) = V(\mathcal{F}, a)$$

typically representing it as a hash table look-up on the atoms in $\mathcal{L}$. For uniformity of notation, let us also write $N_{\mathcal{F}} = N(\mathcal{F})$.

There are several important observations that aid the design of an efficient representation of this computation. First, let us note that:

Lemma 16 *$V_{\mathcal{F}}(a) \subseteq \mathcal{E}$ is a convex space.* □

so it can be represented as a pair of anti-chains. This is done in [2, 3] as the *difference of two upper subsets* of $\mathcal{E}$. The efficiency of this representation owes to the fact that the subsets $V_{\mathcal{F}}(a)$ effectively share a common upper boundary that can be computed in terms of the nogoods. In particular, we can simply maintain the following function mapping atoms to anti-chains in $\mathcal{E}$:

$$L_{\mathcal{F}}(a) = \begin{cases} \min\{x \in \mathcal{E} \mid \mathcal{F} \cup x \models \bot\} & \text{if } a = \bot \\ \min\{x \in \mathcal{E} \mid \mathcal{F} \cup x \not\models \bot \text{ and } \mathcal{F} \cup x \models a\} & \text{if } a \neq \bot \end{cases}$$

Knowing $L_{\mathcal{F}}$ allows us to compute all of the desired values.

Example 17 *If $\mathcal{A} = \{A, B, C, D\}$ is an assumption set and $\mathcal{F}$ is a theory such that $L_{\mathcal{F}}(a) = \{\{A, B\}, \{C\}\}$ and $L_{\mathcal{F}}(\bot) = \{\{C, D\}\}$, then*

$$\begin{aligned} V_{\mathcal{F}}(a) \quad = \quad & \{\{C\}, \{A, B\}, \{A, C\}, \{A, D\}, \{B, C\}, \\ & \{B, D\}, \{A, B, C\}, \{A, B, D\}\} \end{aligned}$$

can be represented as the difference of the upper sets $\uparrow L_{\mathcal{F}}(a)$ and $\uparrow L_{\mathcal{F}}(\bot)$. The convex space $V_{\mathcal{F}}(a)$ is illustrated as the lightly shaded part of Figure 8. □

Lemma 18 *Given a theory $\mathcal{F}$ and assumption set $\mathcal{A}$,*

1. *$N_{\mathcal{F}} = \uparrow L_{\mathcal{F}}(\bot)$.*
2. *$V_{\mathcal{F}}(a) = (\uparrow L_{\mathcal{F}}(a)) - (\uparrow L_{\mathcal{F}}(\bot))$.* □

In practice, it is more efficient to use a *boundary representation* and work with $L_{\mathcal{F}}(\bot)$ and $L_{\mathcal{F}}(a) -^u L_{\mathcal{F}}(\bot)$ than with $N_{\mathcal{F}}$ and $V_{\mathcal{F}}(a)$. These sets are called the *labels* [2] of $\bot$ and a respectively. Knowing the labels is sufficient because we can reconstruct $V_{\mathcal{F}}$ from them:

$$\begin{aligned} \uparrow(L_{\mathcal{F}}(a) -^u L_{\mathcal{F}}(\bot)) - \uparrow L_{\mathcal{F}}(\bot) \quad &= \quad \uparrow(\uparrow L_{\mathcal{F}}(a) - \uparrow L_{\mathcal{F}}(\bot)) - N_{\mathcal{F}} \\ &= \quad \uparrow(\uparrow L_{\mathcal{F}}(a) - N_{\mathcal{F}}) - N_{\mathcal{F}} \\ &= \quad (\uparrow L_{\mathcal{F}}(a)) - N_{\mathcal{F}} \\ &= \quad V_{\mathcal{F}}(a). \end{aligned}$$

So, referring back to our earlier discussion surrounding Lemma 7, the proposed optimization represents the convex space of interest as the difference of two upper sets.

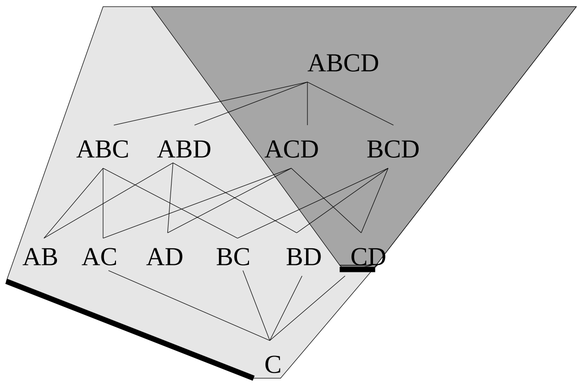

Figure 8: The convex space $V_{\mathcal{F}}(a)$ of environments represented as the difference of two anti-chains $L_{\mathcal{F}}(a) = \{\{A, B\}, C\}$ and $L_{\mathcal{F}}(\bot) = \{\{C, D\}\}$.

Table 3: The ATMS Interface.

`init_atms` : $\text{FinPwr}(\mathcal{L}) \to$ `atms`
`label` : $\mathcal{L} \times$ `atms` $\to \text{Anti}(\mathcal{E})$
`update` : `theory` $\times$ `atms` $\to$ `atms`

The ATMS interface.

An ATMS can be understood as an abstract data type in terms of the semantic interpretation described above. It is used by a client problem-solving system to cache inferences about assumptions that justify propositional atoms. The ATMS data type is defined by the three operators given in Table 3. The ATMS data type itself is denoted `atms` there of course. $\text{FinPwr}(\mathcal{L})$ is the collection of finite sets of propositional atoms. The function `init_atms` creates an ATMS based on a given set of propositional atoms which plays the role of the assumption set $\mathcal{A}$. This function generates an ATMS that associates with each atom $a \in \mathcal{A}$, the label $\{\{a\}\}$, and with each atom $b \in \mathcal{L} - \mathcal{A}$, the empty label. The function `label` computes the label of a propositional atom; this label is an anti-chain in the poset $\mathcal{E}$ of environments (finite subsets of $\mathcal{A}$). The function `update` is the work-horse of the interface: given a theory (set of propositions) and an ATMS, it produces an 'updated' ATMS with labels that incorporate the information provided by the new theory into the labels determined by the previous theories with which the initial ATMS has been updated.

The combination of the basic operations on anti-chains with the basic operations on the ATMS provide the desired functionality for the ATMS in a modular, semantically clear, and efficiently implementable manner. For instance, the functions in [2] that the ATMS is meant to compute can all be succinctly expressed using our interfaces. The same is true for most of the functions in [4]. To see some examples from the latter, consider the function `node-consistent-with?` from [4] (page 440). This takes as arguments an atom $a \in \mathcal{L}$ and an environment $x \in \mathcal{E}$; it returns the boolean value true if, in the label of a, there is a consistent environment y that is a superset of x. In other words, x has a (consistent) extension to an environment from which a follows. In terms of our ATMS and anti-chain interfaces (that is, in terms of the operators in Tables 3, 11, and 12), this is simply

$$\texttt{lower\_member}(\texttt{label}(a, \texttt{theATMS}), x)$$

where `theATMS` is the ATMS of interest. $\texttt{upper\_member}(A, E)$ checks if element E is a member of the upper set corresponding to the anti-chain A. Another example, described in [4] as useful for 'sophisticated inference engines' is a function `supporting-antecedent?` which takes a set of atoms $S = \{a_1, \ldots, a_n\}$ and an environment x as arguments; it checks whether the conjunction of the atoms in S holds in x. In terms of our interfaces, this is $\texttt{upper\_member}(S', x)$ where, with some mixing of mathematical and programming notation,

$$S' = \texttt{label}(a_1, \texttt{theATMS}) \cap^u \cdots \cap^u \texttt{label}(a_n, \texttt{theATMS}).$$

Using the binary operation `upper_homogeneous_intersection`, the anti-chain S' is calculated by using the analog of the Common Lisp utility `apply`.

We can extend our interface to provide additional information to a client problem solver. For example, we can include a function

$$\texttt{atoms} : \texttt{atms} \to \mathrm{FinPwr}(\mathcal{L})$$

which returns the (finite) set of atoms mentioned in the current theory of the ATMS. This would make it possible to ask for the collection of atoms that are 'believed' relative to a given environment x. This collection is sometimes called the *context* of x; it can be calculated by collecting all of the atoms a in $\texttt{atoms}(\texttt{theATMS})$ such that x is a superset of an element of $L = \texttt{label}(a, \texttt{theATMS})$. In particular, x is a superset of such an element holds just in case $\texttt{upper\_member}(L, x)$ is equal to true. However, it seems unlikely that one will want to actually form the context of an environment as a set; in most cases one will want to know only whether an atom is in the context determined by a given environment, and this can be learned from the functions `label` and `upper_member` without recourse to `atoms`. On the other hand, an implementation of the ATMS requires the function `atoms`, so there is no cost in making it available to the client problem solver.

To provide the function `explain-node` described in [4] requires an extension of a different nature to our basic interface. This function (described on page 442 of [4]) takes an atom and an environment as arguments; it returns a proof for the atom based on the theory of the ATMS and the given environment. For our present formulation of the ATMS we have chosen not to keep any information about proofs of atoms from assumptions, instead retaining only the anti-chain of minimal sets of assumptions from which a atom can be derived. To add this function, we need to provide a propositional theorem prover that can reconstruct the proofs of the atoms from the anti-chain of assumptions associated with it. The design of appropriate interfaces for the ATMS is a very interesting and rich subject in its own right; the appropriateness of various choices are essentially compromises between the needs of the client problem-solving system and the computational overhead of delivering the needed functionality.

The basic ATMS.

While we have imposed no restrictions on the theories that can be used with an ATMS, in the absence of such restrictions the `update` function may involve computation that is exponential in the size of $\mathcal{A}$. One way to ensure that `update` can be calculated efficiently for a reasonably expressive language is to restrict the propositions in theories to be Horn clauses. A *Horn clause* is a proposition of the form

$$a_1 \wedge \cdots \wedge a_n \Rightarrow a$$

where $n = 0$ means the proposition is just a, which we may write in the form $\Rightarrow a$. For many applications this has proven to provide a good balance between logical expressiveness and efficiency. The restriction to Horn clauses was used in de Kleer's original paper [2], as well as in much of the discussion of the ATMS in [4]. The computational advantage to Horn clauses lies in the fact that the complexity of the decision problem $\mathcal{F} \models a$ is linear in the size of $\mathcal{F}$ (as opposed to exponential in the size of $\mathcal{A}$).

The logical meaning of the interface function `update` was specified to be the label function $L_{\mathcal{F}}$ where $\mathcal{F}$ is the theory associated with the

ATMS. The question still remains of how this function is to be *calculated.* Our goal now is to provide an abstract description of how this can be done in terms of anti-chains using a class of functions called 'closure operators'.[2] The idea is to do this as a declarative 'executable' specification. In other words, the computation is described as a mathematical entity: in particular, as the least fixed-point of a functional as one does in the denotational semantics of programming languages [12, 6]. The benefit of a mathematical treatment can be realized in proving correctness of algorithms for computing labels. Indeed a proof of correctness based on a fixed-point semantics was given in [5]. What we add to their treatment (as far as the basic ATMS is concerned) is a succinct formulation in terms of our anti-chain operations together with a fuller discussion of the role of closure operators. The use of anti-chain operators makes it possible to capture the mathematical description in pseudo-code that is quite close to an actual efficient implementation one might use. We first show how properties of closure operators allow one to describe the label update algorithm for the basic ATMS; in a later section we use them to develop a new algorithm for the ATMS relative to a more general class of propositions.

Turning now to the technical details, let Anti($\mathcal{E}$) be the set of anti-chains over $\mathcal{E}$. Let us form a partial ordering of anti-chains by taking $x \preceq y$ if, and only if, $\uparrow x \subseteq \uparrow y$. Suppose ϕ is a Horn clause $a_1 \wedge \cdots \wedge a_n \Rightarrow a$; the proposition ϕ induces a functional which can be viewed as an operation for 'improving' the information in a label for the atom a. Repeated application of such information-improving functionals is the key idea of the ATMS label update algorithm. Suppose that F is a function from $\mathcal{L}$ to Anti($\mathcal{E}$). We can view F as describing the 'current state of knowledge' about the label function. Let us say that another such function G is 'at least as informative' as F and write $F \preceq G$ if $F(b) \preceq G(b)$ for each atom b. Taking into account the information provided by ϕ is done by applying an operator

$$\hat{\phi} : (\mathcal{L} \to \mathrm{Anti}(\mathcal{E})) \to (\mathcal{L} \to \mathrm{Anti}(\mathcal{E}))$$

which is defined by the equation

$$\hat{\phi}(F)(b) \quad = \quad \begin{cases} F(b) & \text{if } b \neq a \\ F(a) \cup^u \bigcap^u \{F(a_i) \mid 1 \leq i \leq n\} & \text{if } b = a. \end{cases}$$

There are three special characteristics of $\hat{\phi}$ whose proof we discuss below. First of all, if G is a labeling function that is at least as informative as

[2]The mathematical notion of a closure operator should not be confused with that of a 'closure' in functional programming languages, where the term refers to a pair consisting of a code pointer and an environment.

F, then $\hat{\phi}(G)$ is at least as informative as $\hat{\phi}(F)$; that is, $F \preceq G$ implies that $\hat{\phi}(F) \preceq \hat{\phi}(G)$. Second, $\hat{\phi}(F)$ contains at least as much information as F; that is, $F(b) \preceq \hat{\phi}(F)(b)$ for any atom b. Third, applying $\hat{\phi}$ twice consecutively adds no further information; that is, $\hat{\phi}(\hat{\phi}(F)) = \hat{\phi}(F)$. These are the defining properties of a closure operator:

Definition: Let P be a poset and suppose $f : P \to P$ is a function. Then f is a *closure operator* if it is

- *Monotone:* $x \preceq y$ implies $f(x) \preceq f(y)$,
- *Inflationary:* $x \preceq f(x)$ for any $x \in P$, and
- *Idempotent:* $f(f(x)) = f(x)$. □

Lemma 19 *For any Horn clause ϕ, the function $\hat{\phi}$ is a closure operator.*

Proof: Say ϕ is $a_1 \wedge \cdots \wedge a_n \Rightarrow a$. To see that $\hat{\phi}$ is monotone, suppose $F \preceq G$ and b is an atom. If $b \neq a$, then $\hat{\phi}(F)(b) = F(b) \preceq G(b) = \hat{\phi}(G)(b)$ since $F \preceq G$. On the other hand, if $b = a$, then

$$\begin{aligned}
\uparrow \hat{\phi}(F)(b) &= \uparrow(F(a) \cup^u \textstyle\bigcap^u \{F(a_i) \mid 1 \leq i \leq n\}) \\
&= (\uparrow F(a)) \cup \textstyle\bigcap \{\uparrow F(a_i) \mid 1 \leq i \leq n\}) \\
&\subseteq (\uparrow G(a)) \cup \textstyle\bigcap \{\uparrow G(a_i) \mid 1 \leq i \leq n\}) \\
&= \uparrow(G(a) \cup^u \textstyle\bigcap^u \{G(a_i) \mid 1 \leq i \leq n\}) \\
&= \uparrow \hat{\phi}(G)(b)
\end{aligned}$$

so $\hat{\phi}$ is monotone.

To see that it is inflationary, suppose $b \neq a$, then $F(b) = \hat{\phi}(F)(b)$. If $b = a$, then

$$\begin{aligned}
\uparrow \hat{\phi}(F)(b) &= \uparrow(F(a) \cup^u \textstyle\bigcap^u \{F(a_i) \mid 1 \leq i \leq n\})) \\
&= (\uparrow F(a)) \cup \textstyle\bigcap \{\uparrow F(a_i) \mid 1 \leq i \leq n\})) \\
&\supseteq \uparrow F(b).
\end{aligned}$$

To prove idempotence, we consider two cases. Either $a = a_i$ for some i or $a \neq a_i$ for all i. If the former holds then ϕ is 'uninformative' and it is easy to verify that $\hat{\phi}(F) = F$ for any F. If the latter holds, then $\hat{\phi}(F)(a_i) = F(a_i)$ for each i and we can calculate as follows for the case in which the argument is a:

$$\begin{aligned}
\hat{\phi}(\hat{\phi}(F))(a) &= \hat{\phi}(F)(a) \cup^u \textstyle\bigcap^u \{\hat{\phi}(F)(a_i) \mid 1 \leq i \leq n\} \\
&= (F(a) \cup^u \textstyle\bigcap^u \{F(a_i) \mid 1 \leq i \leq n\}) \cup^u \\
&\quad \textstyle\bigcap^u \{\hat{\phi}(F)(a_i) \mid 1 \leq i \leq n\} \\
&= F(a) \cup^u \textstyle\bigcap^u \{F(a_i) \mid 1 \leq i \leq n\}) \\
&= \hat{\phi}(F)(a)
\end{aligned}$$

If the argument is $b \neq a$, then the result is immediate, so we are done. $\square$

The key point concerning closure operators and partial information is that the repeated application of members of a family S of such operators eventually leads to a point of stability in which no new information is added by additional applications of operators from S. This point of stability is technically a least common fixed point of the operators in S. We require a slightly more general fact which the next theorem expresses.

Theorem 20 *Suppose P is a finite poset and S is a family of closure operators $f : P \to P$. For any point $x_0 \in P$, there is a least common fixed point of S above x_0. That is, there is an element $x \in P$ such that $x_0 \preceq x$ and $f(x) = x$ for each $f \in S$.*

This is a corollary of the following Lemma, which describes how such a fixed point can be computed.

Lemma 21 *Suppose that $\{f_i \mid i \in I\}$ is a family of monotone and inflationary functions on a poset P indexed by some finite set I. Suppose that P is finite and $x_0 \in P$. Let σ be an infinite sequence of members of I such that every element of I appears in every suffix of σ. Let $\sigma[n]$ be the n'th element of σ. Consider the set*

$$\Gamma = \{x_0,\ f_{\sigma[1]}(x_0),\ f_{\sigma[2]}(f_{\sigma[1]}(x_0)),\ \ldots\}.$$

This set has a least upper bound x. This point x is the least common fixed point above x_0 for the functions $\{f_i \mid i \in I\}$.

Proof: The fact that Γ has a least upper bound follows from the observation that it is a chain (that is, $y \preceq z$ or $z \preceq y$ for each $y, z \in \Gamma$) and the poset P is finite.

To prove that x is a common fixed point, we need a convenient notation for composing sequences of f_i's. We write σ_n for the length n prefix of σ. We write f_{σ_n} for the composition $f_{\sigma[n]} \circ f_{\sigma[n-1]} \circ \cdots \circ f_{\sigma[1]}$. First, note that the elements of Γ are written in non-decreasing order because all the f_i are inflations. Second, because P is finite, the least upper bound is attained at some finite stage, that is, for some n, the least upper bound of Γ is $f_{\sigma[n]}(f_{\sigma[n-1]}(\cdots(f_{\sigma[1]}(x_0))))$ or, using our notation, $f_{\sigma_n}(x_0)$. Let us begin by noting that

$$\text{if } m \geq n, \text{ then } x = f_{\sigma_m}(x) \qquad (7)$$

Since x is the least upper bound of Γ, we already know that $f_{\sigma_m}(x_0) \preceq x$, and since $f_{\sigma[n+1]}, \ldots, f_{\sigma[k]}$ are all inflations, we also know that $x = f_{\sigma_n}(x_0) \preceq f_{\sigma_m}(x_0)$.

Now, consider one of the functions f_i. We must show that $f_i(x) = x$. To this end, let m be a number greater than n such that $\sigma[m] = i$; we know there must be such an m from the assumption that i appears in every suffix of σ. By Display (7), we know that $x = f_{\sigma_{m-1}}(x_0)$. Applying $f_{\sigma[m]}$ to both sides of this equation yields $f_i(x) = f_{\sigma_m}(x_0)$, which, by Display (7) again, is equal to x. Thus, x is a fixed point of any of the functions in Γ.

Suppose that x' is any other common fixed point, we have, by an easy induction on m, that for all m, $f_{\sigma_m}(x_0) \preceq x'$ by the monotonicity of the f_i and the fact that $x_0 \preceq x'$. Thus, x' is an upper bound for Γ and, since x is the least upper bound, $x \preceq x'$. □

The upshot of this lemma is that if we wish to find the least common fixed point of a set of closure operators we need only apply each one often enough in succession, not necessarily in any systematic order, and we will find the fixed point. The appearance of the infinite sequence σ in the proof is only to formalize the notion of 'often enough'; the fact that P is finite ensures that only finitely many iterations of such applications are required in order to reach the desired fixed point.

We now use this theorem to express the ATMS label update computation. Suppose that $L_{\mathcal{G}}$ is the label function constructed from the set $\mathcal{G}$ of Horn clauses. The improvement of the label function $L_{\mathcal{G}}$ by a set $\mathcal{F}$ of Horn formulas is characterized as follows.

Theorem 22 (Soundness of the Basic ATMS Algorithm) *Let $\mathcal{A}$ be a set of assumption atoms, and suppose $\mathcal{E}$ is the associated environment lattice. Let L be the label function for $\mathcal{A}, \mathcal{E}$, and suppose that $\mathcal{F}$ and $\mathcal{G}$ are sets of Horn clauses. If $L_{\mathcal{G}} : \mathcal{L} \to \mathrm{Anti}(\mathcal{E})$ is the label function relative to $\mathcal{G}$, then there is a least common fixed point F of the functions*

$$\{\hat{\phi} \mid \phi \in \mathcal{F} \cup \mathcal{G}\}$$

above $L_{\mathcal{G}}$. Moreover, this fixed point satisfies the equation:

$$L_{\mathcal{F}\cup\mathcal{G}}(a) \quad = \quad \begin{cases} F(a) -^{u} F(\bot) & \text{if } a \neq \bot \\ F(\bot) & \text{if } a = \bot \quad \Box \end{cases} \tag{8}$$

A proof of the theorem involves relating the semantic entailment $\models$ for the Horn clause formulas ϕ to the least fixed point of the closure operators $\hat{\phi}$. This can be done through the use of a *minimal* model for a collection of Horn clauses. Details sufficient to construct a proof of Theorem 22 can be found in [5] and [11].

Table 4: Converting a Horn Clause ϕ to a Closure Operator $\hat{\phi}$.

```
function $\hat{\phi}$(F)(b) =
  if b $\neq$ a then F(b)
  else F(a) $\cup^u$ (fold($\cap^u$, map(F)(S), $\{\emptyset\}$))
  endif
```

where ϕ is a Horn clause such that S is the set of its premises and `a` is its conclusion.

It is worth noting how Theorem 22 reflects the 'non-monotonicity' of labels despite using a fixed point computation based on monotonic operators. Since some members of $L_{\mathcal{G}}(a)$, for $a \neq \bot$, may become nogoods when deductions based on $\mathcal{F}$ are taken into account, it is not necessarily the case that $L_{\mathcal{G}}(a) \preceq L_{\mathcal{F}\cup\mathcal{G}}(a)$. Put more generally: although F is a monotone operator on labeling functions, $L_X(a)$ may not be a monotone function of X (under an ordering of sets of formulas by subset inclusion). This non-monotonicity arises from the upper difference $F(a) -^u F(\bot)$ taken in Display 8.

Using Lemma 21, Display 8 shows how to calculate $L_{\mathcal{F}\cup\mathcal{G}}$ from $L_{\mathcal{G}}$ and $\mathcal{F}$ by the use of a least common fixed point. We can convert this mathematics for ATMS computation into pseudo-code in the way Tables 1 and 2 did for version spaces. Pseudo-code for the operation $\phi \mapsto \hat{\phi}$, which takes a Horn clause to the corresponding closure operator on label functions, is given in Table 4. In the program there, the following functions are taken from the anti-chains interface:

`upper_union`	`upper_homogeneous_intersection`
`singleton`	`empty`

The first two of these we have encountered before. The operation `singleton` takes an element x and forms the singleton anti-chain $\{x\}$. `empty` is the empty set, which is itself an anti-chain. In particular, the upper set that the anti-chain `singleton(empty)` represents is $\uparrow\{\emptyset\} = \mathcal{E}$. Aside from the operations on anti-chains, we require two basic operations on sets:

`map`	`fold`

The function `map` takes two arguments, a function F and a set S; it returns the set obtained by applying F to each of the elements of S.

Table 5: Basic ATMS Label Update.

```
function improveLABEL(L) =
  let L' = C(L)
  in  if L = L'
      then $\lambda$(atom) =>
             if atom = $\perp$ then L($\perp$)
             else L(atom) $-^u$ L($\perp$)
      else improveLABEL(L')
      endif
  endlet
```

for $\texttt{C} = \hat{\phi}_1 \circ \cdots \circ \hat{\phi}_n$ where $\mathcal{F} \cup \mathcal{G} = \{\phi_1, \ldots, \phi_n\}$.

For instance `map square` $\{1, 2, 3\} = \{1, 4, 9\}$. A Common Lisp analog for lists is `mapcar`. The function `fold` takes a binary operation $*$, a set $T = \{x_1, \ldots, x_n\}$, and an 'end value' x; it returns $x_1 * x_2 * \cdots * x_n * x$. If $T = \emptyset$, then `fold` $*\ T\ x = x$. For instance `fold` $+\ \{1, 2, 3\}\ 0 = 6$ and `fold` $+\ \emptyset\ \ 42 = 42$. The Common Lisp function `apply` is similar but does not use the 'end value' x. In Table 4, the end value used is $\{\emptyset\}$ (which is not to be confused with the empty set itself). In particular, note the case in which ϕ has the form $\Rightarrow a$ so that S is the emptyset. In this case an upper union is taken between $F(a)$ and the singleton containing empty set. The upper union of these is the singleton containing the empty set. The upper set of this is the entire environment lattice; this is what we expect, since a is a premise and therefore holds in every environment.

To use the operation $\phi \mapsto \hat{\phi}$ to calculate a label one can use a 'label improvement' function; pseudo-code for a such a function appears in Table 5. Since the result of evaluating `improveLABEL(L)` should be a new labeling function, the returned value must itself be a function. This is represented in the pseudo-code using a form

```
$\lambda$(x) => $E$
```

which is an 'anonymous' function with formal parameter `x` and body E.

The algorithms in Tables 4 and 5 correctly implement the ATMS computation. However, they are less efficient than they could be: at least three changes can be used to optimize the algorithms for some cases.

1. *Improving the termination test.* The test `L' = L` in the third line of Table 5 can be combined with the calculation of `C(L)`, saving a

second pass through the structures `L` and `L'`. In an actual implementation, this is accomplished by using a flag that is set whenever a change to `L` occurs. The test `CL` = `L` is then replaced by a test to check if this flag is set.

2. *Optimizing the choice of application sequences of closure operators.* It is possible to determine as the computation proceeds that some of the closure operators will not yield new information in the current state of information. In particular, when the label of an atom a is updated, the only 'directly relevant' Horn clauses to consider are those in which a occurs as an antecedent.

3. *Incrementally removing nogoods.* Rather than removing nogoods at end of the calculation as in `improveLABEL`, they can be eliminated incrementally during the course of an update, as new environments are added to the anti-chain of nogoods.

All these optimizations are found in real implementations of basic ATMS update algorithm. The Forbus-deKleer algorithm from [4] is reconstructed in Table 6 using our anti-chain operations. The closure operators that occur in the declarative specification of the ATMS are implemented in Table 6 using procedural iteration constructs.

The chief insight in the algorithm is to propagate *changes* to labels rather than the entire labels in the update computation. We can define the *incremental* information provided by a Horn clause ϕ as the application of an operator

$$\Delta\hat{\phi} : (\mathcal{L} \to \mathrm{Anti}(\mathcal{E})) \to (\mathcal{L} \to \mathrm{Anti}(\mathcal{E}))$$

which is defined by the equation

$$\Delta\hat{\phi}(F)(b) \quad = \quad \begin{cases} \emptyset & \text{if } b \neq a \\ \bigcap^u \{F(a_i) \mid 1 \leq i \leq n\} & \text{if } b = a \end{cases}$$

The procedure `WEAVE` in Table 6 implements $\Delta\hat{\phi}$. Nogoods are removed as they are discovered in the deKleer-Forbus algorithm. The procedure `NOGOOD` in Table 6 implements incremental nogood removal by in-place update of the evolving label function F. When an antichain S is added to the evolving anti-chain of nogoods $F(\perp)$, then every label set $F(a)$, $a \neq \perp, a \in \mathcal{L}$ is updated to eliminate environments in S and those subsumed by S (i.e., $F(a) -^u S$). In procedure `UPDATE`, the label of an atom `a` is updated to include the new labels in the anti-chain `S`. In the `for` loop in this procedure the closure operator consisting of the "relevant" Horn clauses (ones in which `a` occurs in the antecedent) is constructed

Table 6: Basic ATMS Label Update Algorithm.

Assume ϕ is a Horn clause of the form $a_1, \ldots, a_n \Rightarrow a$.

```
function PROPAGATE ($\phi$, b, I) =
  let S = WEAVE(b, I, $\{a_1, \ldots, a_n\}$)
   in if S $\neq$ $\emptyset$ then UPDATE(S, a) endif
  endlet

function UPDATE(S, a) =
  if a = $\perp$
  then  NOGOOD(S)
  else L(a) := S $\cup^u$ L(a)
       for each $\phi$ such that a is an antecedent of $\phi$
          do PROPAGATE($\phi$,a,S);
            S := [S $\cup^u$ L(a)] $-^u$ L($\perp$);
            if S is empty then return endif
       endfor
  endif;

function WEAVE(b, I, $\{a_1, \ldots, a_n\}$) =
  for each $a_i$
   do if $a_i$ $\neq$ b then I := (I $\cap^u$ L($a_i$)) $-^u$ L($\perp$) endif
  endfor;
  return I

function NOGOOD(S) =
  L($\perp$) := L($\perp$) $\cup^u$ S;
  for each atom b $\neq$ $\perp$
   do L(b) := L(b) $-^u$ S
  endfor
```

incrementally and the label function F updated by each of these Horn clauses in turn (by procedure `PROPAGATE`). `PROPAGATE` computes $\Delta\hat{\phi}$ with respect to a given Horn clause (by using `WEAVE`) and terminates label propagation if there is no change in the label set ($S = \emptyset$). The compact semi-procedural/declarative reconstruction of the optimizations in this algorithm allows us to use the anti-chains package for effective implementation.

5 Extended ATMS's

Closure operators have a crucial but largely unappreciated importance for designing algorithms that manipulate partial information. We now illustrate this for what de Kleer called the 'extended ATMS'.

The extended ATMS is defined in [3] to be an ATMS where the input theory consists of Horn clauses over $\mathcal{L}$ and disjunctions of assumptions $A_1 \vee \ldots \vee A_n$. The extended ATMS has the same interface (Table 3) as the basic ATMS, but permits certain kinds of disjunctions to be given in theories, in addition to Horn clauses. That is, the goal is to compute $L_{\mathcal{F}\cup\mathcal{T}}(a)$ for propositional atoms $a \in \mathcal{L}$, where $\mathcal{F}$ is a set of Horn clauses and $\mathcal{T}$ is a set of disjunctions of assumptions.

Example 23 *If $\mathcal{A} = \{A, B, C\}$ is an assumption set,*

$$\mathcal{F} = \{A \Rightarrow a,\ B \Rightarrow b,\ C \Rightarrow c,\ c \wedge a \Rightarrow \bot,\ c \wedge b \Rightarrow \bot\}$$

is a set of Horn clauses, and $\mathcal{T} = \{A \vee B\}$ is a set of disjunctions of assumptions, then

$$L_{\mathcal{F}}(\bot) = \{\{A, C\}, \{B, C\}\}$$
$$L_{\mathcal{F}\cup\mathcal{T}}(\bot) = \{\{C\}\}$$

since one of A or B is guaranteed to hold in models of $\mathcal{F} \cup \mathcal{T}$. □

As de Kleer notes in [3], the label update algorithm for the basic ATMS, which is sound and complete for Horn clause theories, becomes *in*complete when disjunctions are allowed. To correctly compute labels with respect to $\mathcal{F}\cup\mathcal{T}$, [3] uses two hyper-resolution rules to 'fix up' labels computed with respect to $\mathcal{F}$ by the basic ATMS algorithm.

Our approach to an extended ATMS generalizes the disjunctions of assumptions used in [3], to formulas of the form:

$$\tau = \phi_1 \vee \phi_2 \vee \cdots \vee \phi_n$$

where each ϕ_i has the form

$$\phi_i = A_1^i \wedge \cdots \wedge A_{n_i}^i.$$

and $A_j^i \in \mathcal{A}$.

In other words, simple disjunctions of assumptions are generalized to formulas in disjunctive normal form over assumptions. Let us refer to a formula having the form of τ as an *Assumption Disjunctive Normal Form (ADNF).* To appreciate why generalizing from primitive disjunctions to ADNF's is a natural step, note that a conjunction of assumptions like ϕ_i

is essentially equivalent to an environment $\{A^i_1, \ldots, A^i_{n_i}\} \in \mathcal{E}$. A formula like τ can therefore be viewed as an *anti-chain* over $\mathcal{E}$, provided the ϕ_i's are not subsumed by one another. In effect, we are permitting the logical import of an anti-chain to be directly asserted by the problem-solving system in the form of an ADNF. To develop the details it will help us to confuse the distinction between an ADNF like τ and a set of environments. So we write $t \in \tau$ to mean that $t = \{A^i_1, \ldots, A^i_{n_i}\}$ for some $i \leq n$. That is, t is the environment that corresponds to ϕ_i. We use symbols $\mathcal{T}, \mathcal{T}'$ for sets of ADNF formulas. Our mathematical treatment in terms of closure operators allows us to derive a new algorithm for computing labels with respect to theories that include assumption DNF's and to prove its soundness. In addition, new optimizations in label computations that were otherwise hidden are revealed in this formulation.

We now examine how the introduction of assumption DNF formulas over $\mathcal{A}$ changes the set of environments in which a propositional atom holds. Continuing with Example 23:

$$\begin{aligned} N_{\mathcal{F}} = \uparrow L_{\mathcal{F}}(\bot) &= \{\{A, C\}, \{B, C\}, \{A, B, C\}\} \\ N_{\mathcal{F}\cup\mathcal{T}} = \uparrow L_{\mathcal{F}\cup\mathcal{T}}(\bot) &= \{\{C\}, \{A, C\}, \{B, C\}, \{A, B, C\}\} \end{aligned}$$

The set of environments in which $\bot$ holds expands with the introduction of the disjunction $A \vee B$ to include environments $x \in \mathcal{E}$ such that $x \cup \{A\} \in N_{\mathcal{F}}$ and $x \cup \{B\} \in N_{\mathcal{F}}$. To identify such environments added by the assumption DNF formula $\tau \in \mathcal{T}$, we define the operator

$$\Psi_\tau : \mathrm{Pwr}(\mathcal{E}) \rightarrow \mathrm{Pwr}(\mathcal{E})$$

which is intended to extend the upward closed set of environments $V_{\mathcal{F}}(a)$ and $N_{\mathcal{F}}$ to the upward closed sets $V_{\mathcal{F}\cup\{\tau\}}(a)$ and $N_{\mathcal{F}\cup\{\tau\}}$ respectively. Given a set $S \subseteq \mathcal{E}$, define

$$\Psi_\tau(S) = \{x \in \mathcal{E} \mid \forall t \in \tau.\ x \cup t \in \uparrow S\}$$

It is easy to see that $\Psi_\tau(S)$ is an upward closed set, i.e., $\uparrow \Psi_\tau(S) = \Psi_\tau(S)$. For instance, given the theories defined in Example 23:

$$N_{\mathcal{F}\cup\{A\vee B\}} = \Psi_{\{\{A\},\{B\}\}}(N_{\mathcal{F}}).$$

Note also that $\Psi(S) = \Psi(\uparrow S)$ for any S. Ultimately, however, we are interested in working with operations on *anti-chains* of environments. We define the anti-chain analog

$$\Phi_\tau : \mathrm{Anti}(\mathcal{E}) \rightarrow \mathrm{Anti}(\mathcal{E})$$

of Ψ_τ as follows:

$$\Phi_\tau(S) = \min(\Psi_\tau(S))$$

or, equivalently, $\Phi_\tau(S)$ is the unique anti-chain in $\mathcal{E}$ such that

$$\uparrow \Phi_\tau(S) = \Psi_\tau(S).$$

Lemma 24 *The function Ψ_τ is a closure operator for any ADNF formula τ.*

Proof: Let $S \subseteq T$, where $S, T \in \mathrm{Pwr}(\mathcal{E})$. This means that $\uparrow S \subseteq \uparrow T$. We first establish monotonicity.

$$\begin{aligned} x \in \Psi_\tau(S) &\Rightarrow \forall t \in \tau.\ x \cup t \in \uparrow S \\ &\Rightarrow \forall t \in \tau.\ x \cup t \in \uparrow T \\ &\Rightarrow x \in \Psi_\tau(T). \end{aligned}$$

To see that Ψ_τ is inflationary, suppose $x \in S$ and $t \in \tau$. Then clearly $x \cup t \subseteq \uparrow S$, so $x \in \Psi_\tau(S)$. Thus $S \subseteq \Psi_\tau(S)$. To demonstrate idempotence, we show that $\Psi_\tau(\Psi_\tau(S)) = \Psi_\tau(S)$. We know that $\Psi_\tau(S) \subseteq \Psi_\tau(\Psi_\tau(S))$. To prove the opposite inclusion, let x be an environment:

$$\begin{aligned} x \in \Psi_\tau(\Psi_\tau(S)) &\Rightarrow \forall t \in \tau.\ x \cup t \in \uparrow \Psi_\tau(S) \\ &\Rightarrow \forall t \in \tau.\ x \cup t \in \Psi_\tau(S) \\ &\Rightarrow \forall t \in \tau.\ x \cup t \in \{x' \in \mathcal{E} \mid \forall t' \in \tau.\ x' \cup t' \in \uparrow S\} \\ &\Rightarrow \forall t \in \tau.\ x \cup t \cup t \in \uparrow S \\ &\Rightarrow x \in \Psi_\tau(S). \end{aligned}$$

Hence Ψ_τ is idempotent and therefore a closure operator. □

Corollary 25 *The function Φ_τ is a closure operator for any ADNF formula τ.*

Proof: Recall that $S \preceq T$ for anti-chains S, T means $\uparrow S \subseteq \uparrow T$. Now,

$$\uparrow \Phi_\tau(S) = \Psi_\tau(S) \subseteq \Psi_\tau(T) = \uparrow \Phi_\tau(T)$$

so Φ_τ is monotone. Moreover, $\uparrow S \subseteq \Psi_\tau(\uparrow S) = \Psi_\tau(S)$ means

$$S = \min(\uparrow S) \preceq \min(\Psi_\tau(S)) = \Phi_\tau(S)$$

so Φ_τ is inflationary. Finally,

$$\uparrow \Phi_\tau(\Phi_\tau(S)) = \Psi_\tau(\Phi_\tau(S)) = \Psi_\tau(\uparrow \Phi_\tau(S)) = \Psi_\tau(\Psi_\tau(S))$$

so $\Phi_\tau(\Phi_\tau(S)) = \Phi_\tau(S)$ too. Hence Φ_τ is idempotent. □

In general, results like this corollary follow from the fact that Φ and Ψ are corresponding operators on isomorphic spaces (that is, on anti-chains and upper sets respectively).

Lemma 26 *Suppose* $S \in \mathrm{Anti}(\mathcal{E})$ *and* τ_1, τ_2 *are ADNF's. Then*

$$\Phi_{\tau_1}(\Phi_{\tau_2}(S)) = \Phi_{\tau_2}(\Phi_{\tau_1}(S)) = \Phi_{\tau}(S),$$

where $\tau = \min\{x \in \mathcal{E} \mid \exists t_1 \in \tau_1 \exists t_2 \in \tau_2.\ x = t_1 \cup t_2\}$.

Proof: Let S be an anti-chain of environments and suppose y is an environment, then

$$\begin{aligned}
& y \in \Phi_{\tau_1}(\Phi_{\tau_2}(S)) \\
\Leftrightarrow\ & \forall t_1 \in \tau_1.\ y \cup t_1 \in \uparrow \Phi_{\tau_2}(S) \\
\Leftrightarrow\ & \forall t_1 \in \tau_1\ \exists z_1 \in \Phi_{\tau_2}(S).\ z_1 \subseteq y \cup t_1 \\
\Leftrightarrow\ & \forall t_1 \in \tau_1\ \exists z_1.\ (\forall t_2 \in \tau_2.\ z_1 \cup t_2 \in \uparrow S) \text{ and } z_1 \subseteq y \cup t_1 \\
\Leftrightarrow\ & \forall t_1 \in \tau_1\ \forall t_2 \in \tau_2.\ (y \cup t_1) \cup t_2 \in \uparrow S \\
\Leftrightarrow\ & y \in \Phi_{\tau}(S) \\
\Leftrightarrow\ & \forall t_2 \in \tau_2\ \forall t_1 \in \tau_1.\ (y \cup t_2) \cup t_1 \in \uparrow S \\
\Leftrightarrow\ & \forall t_2 \in \tau_1\ \exists z_2.\ (\forall t_1 \in \tau_1.\ z_2 \cup t_1 \in \uparrow S) \text{ and } z_2 \subseteq y \cup t_2 \\
\Leftrightarrow\ & \forall t_2 \in \tau_1\ \exists z_2 \in \Phi_{\tau_1}(S).\ z_2 \subseteq y \cup t_2 \\
\Leftrightarrow\ & \forall t_2 \in \tau_2.\ y \cup t_2 \in \uparrow \Phi_{\tau_1}(S) \\
\Leftrightarrow\ & y \in \Phi_{\tau_2}(\Phi_{\tau_1}(S)) \qquad \square
\end{aligned}$$

It is illuminating to note that the formula τ in this Lemma essentially corresponds to the anti-chain that one could obtain from taking an upper intersection of τ_1 and τ_2 viewed as anti-chains, that is $\tau = \tau_1 \cap^u \tau_2$.

Lemma 27 *For* $S \in \mathrm{Anti}(\mathcal{E})$, *and for closure operators* $\Phi_{\tau_1}, \ldots, \Phi_{\tau_n}$, *the least common fixed point of* $\{\Phi_{\tau_i} \mid 1 \leq i \leq n\}$ *above* S *is* $\Phi_{\tau_1}(\Phi_{\tau_2}(\ldots(\Phi_{\tau_n}(S)\ldots)))$.

Proof: This is a consequence of Corollary 25, Lemma 26, and Lemma 21. $\square$

Given a set $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ of ADNF formulas, let us define $\Phi_{\mathcal{T}}(S)$ to be the least fixed point above S of the closure operators $\Phi_{\tau_1}, \ldots, \Phi_{\tau_n}$. Lemma 27 tells us how to compute $\Phi_{\mathcal{T}}(S)$.

Our goal now is to show how to calculate the label function for the extended ATMS. This is done in two parts: first, we show how to compute labels using the operators $\Phi_{\mathcal{T}}$ and, second, we show how to compute $\Phi_{\mathcal{T}}(S)$ for anti-chains S. For the first step, here is the desired result:

Theorem 28 Extended ATMS algorithm. *Let $\mathcal{F}$ be a set of Horn clauses over $\mathcal{L}$, and let $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ be a set of ADNF formulae over the assumption set $\mathcal{A}$. Then for every propositional atom $a \in \mathcal{L}$,*

$$L_{\mathcal{F}\cup\mathcal{T}}(a) \quad = \quad \begin{cases} \Phi_{\mathcal{T}}(L_{\mathcal{F}}(a)) & \textit{if } a = \bot \\ \Phi_{\mathcal{T}}(L_{\mathcal{F}}(a) \cup^u L_{\mathcal{F}}(\bot)) -^u L_{\mathcal{F}\cup\mathcal{T}}(\bot) & \textit{if } a \neq \bot \end{cases}$$

The proof of the theorem, which is a demonstration of the soundness of the extended ATMS algorithm, is best done by establishing some general equations as a preliminary.

Let $\mathcal{F}$ be any theory (the formulas in $\mathcal{F}$ do not need to be Horn clauses or ADNF's, although this is the case we are actually interested in). Suppose that $x \in \mathcal{E}$ and τ is an ADNF. Then the following correspondence holds:

$$\mathcal{F} \cup \{\tau\} \cup x \models a \text{ iff } \forall t \in \tau.\ \mathcal{F} \cup t \cup x \models a \tag{9}$$

To see why, let us look first at $\Rightarrow$. Suppose that any model of $\mathcal{F} \cup \{\tau\} \cup x$ is also a model of a, and suppose that $M \models \mathcal{F} \cup t \cup x$ for some $t \in \tau$. Then $M \models \mathcal{F} \cup \{\tau\} \cup x$ too, so by assumption, $M \models a$. Thus $\forall t \in \tau.\ \mathcal{F} \cup t \cup x \models a$. Turning now to a proof of $\Leftarrow$, suppose that for any $t \in \tau$, a model of $\mathcal{F} \cup t \cup x$ is also a model of a. If $M \models \mathcal{F} \cup \{\tau\} \cup x$, then $M \models \tau$, so $M \models t$ for some $t \in \tau$. Hence $M \models \mathcal{F} \cup t \cup x$ and our hypothesis allows us to conclude that $M \models a$ too. This establishes (9).

Now, it will be convenient for us to have a notation for the upper set of environments that prove a given conclusion. So, given a theory $\mathcal{F}$, we define

$$U_{\mathcal{F}}(a) = \{x \in \mathcal{E} \mid \mathcal{F} \cup x \models a\}.$$

Given a theory $\mathcal{F}$ and a collection $\mathcal{T}$ of ADNF formulas, the main fact of interest about U is the following:

$$U_{\mathcal{F}\cup\mathcal{T}} = \Psi_{\mathcal{T}} \circ U_{\mathcal{F}} \tag{10}$$

This can be proved by induction on the number of elements in $\mathcal{T}$. When this is 0, then $\mathcal{T}$ is empty and there is nothing to prove. Suppose $\tau \in \mathcal{T}$ and $\mathcal{T}' = \mathcal{T} - \{\tau\}$ and the desired result is known for $\mathcal{T}'$. Given an atom a,

$$\begin{aligned} U_{\mathcal{F}\cup\mathcal{T}}(a) &= \{x \in \mathcal{E} \mid \mathcal{F} \cup \mathcal{T}' \cup \{\tau\} \cup x \models a\} \\ &= \{x \in \mathcal{E} \mid \forall t \in \tau.\ \mathcal{F} \cup \mathcal{T}' \cup t \cup x \models a\} & (11) \\ &= \Psi_\tau(U_{\mathcal{F}\cup\mathcal{T}'}(a)) \\ &= \Psi_\tau(\Psi_{\mathcal{T}'}(U_{\mathcal{F}}(a))) & (12) \\ &= \Psi_{\mathcal{T}} \circ U_{\mathcal{F}} \end{aligned}$$

Equation (11) follows from (9) and Equation (12) follows from the inductive hypothesis.

Proof: (Of Theorem 28) To calculate $L_{\mathcal{F}\cup\mathcal{T}}(\bot)$, note first that it is defined to be $\min(U_{\mathcal{F}\cup\mathcal{T}}(\bot)))$ which is equal to $\min(\Psi_{\mathcal{T}}(U_{\mathcal{F}}(\bot)))$ by Equation 10. This, in turn, is equal to $\Phi_{\mathcal{T}}(L_{\mathcal{F}}(\bot))$.

If a is an atom other than $\bot$, then

$$\begin{array}{rcl} \Phi_{\mathcal{T}}(L_{\mathcal{F}}(a) \cup^u L_{\mathcal{F}}(\bot)) & = & \min(\Psi_{\mathcal{T}}(\uparrow(L_{\mathcal{F}}(a) \cup^u L_{\mathcal{F}}(\bot)))) \\ & = & \min(\Psi_{\mathcal{T}}(\uparrow L_{\mathcal{F}}(a) \cup \uparrow L_{\mathcal{F}}(\bot))) \\ & = & \min(\Psi_{\mathcal{T}}(U_{\mathcal{F}}(a))) \\ & = & \min(U_{\mathcal{F}\cup\mathcal{T}}(a)) \\ & = & L_{\mathcal{F}\cup\mathcal{T}}(a) \cup^u L_{\mathcal{F}\cup\mathcal{T}}(\bot) \end{array}$$

Thus

$$\begin{array}{rcl} L_{\mathcal{F}\cup\mathcal{T}}(a) & = & (L_{\mathcal{F}\cup\mathcal{T}}(a) \cup^u L_{\mathcal{F}\cup\mathcal{T}}(\bot)) -^u L_{\mathcal{F}\cup\mathcal{T}}(\bot) \\ & = & \Phi_{\mathcal{T}}(L_{\mathcal{F}}(a) \cup^u L_{\mathcal{F}}(\bot)) -^u L_{\mathcal{F}\cup\mathcal{T}}(\bot). \qquad \Box \end{array}$$

What remains is showing how to calculate Φ_τ. Given a collection of anti-chains $S = \{s_1, \ldots, s_n\}$, it will be useful to write

$$\overset{u}{\bigcup_{s\in S}} s = s_1 \cup^u \cdots \cup^u s_n \text{ and } \overset{u}{\bigcap_{s\in S}} s = s_1 \cap^u \cdots \cap^u s_n.$$

The desired expression of $\Phi_\tau(S)$ in terms of anti-chain operations is given by the following:

Lemma 29 *For $S \in \mathrm{Anti}(\mathcal{E})$ and ADNF formula τ over $\mathcal{A}$,*

$$\Phi_\tau(S) = \overset{u}{\bigcap_{t\in\tau}} \overset{u}{\bigcup_{s\in S}} \{s - t\}$$

Proof: We calculate as follows:

$$\begin{array}{rcl} \Phi_\tau(S) & = & \min(\Psi_\tau(S)) \\ & = & \min\{x \in \mathcal{E} \mid \forall t \in \tau.\ x \cup t \in \uparrow S\} \\ & = & \min\{x \in \mathcal{E} \mid \forall t \in \tau.\ \exists s \in S.\ s \subseteq x \cup t\} \\ & = & \overset{u}{\bigcap_{t\in\tau}} \min\{x \in \mathcal{E} \mid \exists s \in S.\ s \subseteq x \cup t\} \\ & = & \overset{u}{\bigcap_{t\in\tau}} \overset{u}{\bigcup_{s\in S}} \min\{x \in \mathcal{E} \mid s \subseteq x \cup t\} \end{array}$$

Now consider the sets $e_t^s = \{x \in \mathcal{E} \mid s \subseteq x \cup t\}$. An environment x is in e_t^s just in case it contains the environment $s - t$. Hence $s - t$ is the unique minimum element of e_t^s. The equation in the lemma therefore follows. $\square$

The algorithms in Table 7 and Table 8 correctly implement the extended ATMS label computations. We now show two examples of the use of these algorithms.

Example 30 We illustrate the computation of the label for $\perp$ in the example introduced at the start of this subsection.

$$\begin{aligned} \mathcal{F} &= \{A \Rightarrow a, B \Rightarrow b, C \Rightarrow c, c \wedge a \Rightarrow \perp, c \wedge b \Rightarrow \perp\} \\ \mathcal{T} &= \{A \vee B\}. \end{aligned}$$

Recall that the assumption set $\mathcal{A} = \{A, B, C\}$. Our aim is to calculate $L_{\mathcal{F} \cup \mathcal{T}}(\perp)$. First, we use the basic ATMS algorithm to compute

$$L_{\mathcal{F}}(\perp) = \{\{A, C\}, \{B, C\}\}$$

Next, the closure operator $\Phi_{\mathcal{T}}$ is constructed from Lemma 29 by the algorithm in Table 7.

$$\Phi_{\{\{A\},\{B\}\}} = [\overset{u}{\bigcup_s}\{s - \{A\}\}] \;\; \cap^u \;\; [\overset{u}{\bigcup_s}\{s - \{B\}\}]$$

Finally, we use Theorem 28 and the algorithm in Table 8 to obtain the desired result.

$$\begin{aligned} & L_{\mathcal{F} \cup \{\{A\},\{B\}\}}(\perp) \\ &= \Phi_{\{\{A\},\{B\}\}}(\{\{A, C\}, \{B, C\}\}) \\ &= [\{\{A, C\} - \{A\}\} \cup^u \{\{B, C\} - \{A\}\}] \\ &\quad \cap^u [\{\{A, C\} - \{B\}\} \cup^u \{\{B, C\} - \{B\}\}] \\ &= [\{\{C\}\} \cup^u \{\{B, C\}\}] \cap^u [\{\{A, C\}\} \cup^u \{\{C\}\}] \\ &= \{\{C\}\} \cap^u \{\{C\}\} \\ &= \{\{C\}\} \end{aligned}$$

Example 31 We demonstrate the power of the anti-chains formulation of label computations in the context of an example with a more complex $\mathcal{T}$. Consider

$$\begin{aligned} \mathcal{F} &= \{A, \overline{A} \Rightarrow \perp, B \Rightarrow b, C \Rightarrow c, b \Rightarrow c, c \Rightarrow d\} \\ \mathcal{T} &= \{\overline{A} \vee B \vee C, A \vee \overline{A}\} \\ &= \{\{\{\overline{A}\}, \{B\}, \{C\}\}, \{\{A\}, \{\overline{A}\}\}\} \end{aligned}$$

We need to calculate $L_{\mathcal{F}\cup\mathcal{T}}(d)$. From the basic ATMS algorithm, we know that

$$L_{\mathcal{F}}(d) = \{\{B\},\{C\}\} \text{ and } L_{\mathcal{F}}(\perp) = \{\{A,\overline{A}\}\}$$

Next, we use Lemma 26 to reduce $\mathcal{T} = \{\tau_1,\tau_2\}$ to a single ADNF formula τ.

$$\begin{aligned}\tau &= \min\{x \in \mathcal{E} \mid \exists t_1 \in \tau_1, \exists t_2 \in \tau_2, x = t_1 \cup t_2\}\\ &= \min\{\{\overline{A},A\},\{\overline{A}\},\{B,A\},\{B,\overline{A}\},\{C,A\},\{C,\overline{A}\}\}\\ &= \{\{\overline{A}\},\{B,A\},\{C,A\}\}\end{aligned}$$

Now we construct $\Phi_{\{\tau\}}$ using the algorithm in Table 7.

$$\begin{aligned}&\Phi_{\{\tau\}}(S)\\ &= [\overset{u}{\bigcup_{s\in S}}\{s-\{\overline{A}\}\}]\\ &\quad \cap^u [\overset{u}{\bigcup_{s}}\{s-\{B,A\}\}]\\ &\quad \cap^u [\overset{u}{\bigcup_{s}}\{s-\{C,A\}\}]\end{aligned}$$

We can then calculate $L_{\mathcal{F}\cup\mathcal{T}}(\perp)$ and $L_{\mathcal{F}\cup\mathcal{T}}(d)$ using the algorithm in Table 8.

$$\begin{aligned}&L_{\mathcal{F}\cup\mathcal{T}}(\perp)\\ &= \Phi_{\mathcal{T}}(\{\{A,\overline{A}\}\})\\ &= [\{\{A,\overline{A}\}-\{\overline{A}\}\}] \cap^u [\{\{A,\overline{A}\}-\{B,A\}\}]\\ &\quad \cap^u[\{\{A,\overline{A}\}-\{C,A\}\}]\\ &= \{\{A\}\} \cap^u \{\{\overline{A}\}\} \cap^u \{\{\overline{A}\}\}\\ &= \{\{A,\overline{A}\}\}\\ &L_{\mathcal{F}\cup\mathcal{T}}(d)\\ &= \Phi_{\mathcal{T}}(L_{\mathcal{F}}(d) \cup^u L_{\mathcal{F}}(\perp)) -^u L_{\mathcal{F}\cup\mathcal{T}}(\perp)\\ &= \Phi_{\mathcal{T}}(\{\{B\},\{C\},\{A,\overline{A}\}\}) -^u \{\{A,\overline{A}\}\}\\ &\Phi_{\mathcal{T}}(\{\{B\},\{C\},\{A,\overline{A}\}\})\\ &= [\{\{B\}-\{\overline{A}\}\} \cup^u \{\{C\}-\{\overline{A}\}\} \cup^u \{\{A,\overline{A}\}-\{\overline{A}\}\}]\\ &\quad \cap^u[\{\{B\}-\{B,A\}\} \cup^u \{\{C\}-\{B,A\}\} \cup^u \{\{A,\overline{A}\}-\{B,A\}\}]\\ &\quad \cap^u[\{\{B\}-\{C,A\}\} \cup^u \{\{C\}-\{C,A\}\} \cup^u \{\{A,\overline{A}\}-\{C,A\}\}]\\ &= [\{\{B\}\} \cup^u \{\{C\}\} \cup^u \{\{A\}\}]\end{aligned}$$

Table 7: Converting an ADNF formula τ to a Closure Operator Φ_τ.

```
function  $\Phi_\tau$(S) =
   fold($\cap^u$,map($\lambda$t. fold($\cup^u$,map($\lambda$s.s - t)(S),$\emptyset$) (T)),{$\emptyset$})
```

where τ is an ADNF formula and S is an anti-chain over $\mathcal{E}$.

$$\begin{aligned}
& \cap^u[\{\emptyset\} \cup^u \{\{C\}\} \cup^u \{\{\overline{A}\}\}] \\
& \cap^u[\{\{B\}\} \cup^u \{\emptyset\} \cup^u \{\{\overline{A}\}\}] \\
= \ & \{\{B\},\{C\},\{A\}\} \\
& L_{\mathcal{F}\cup\mathcal{T}}(d) \\
= \ & \{\{B\},\{C\},\{A\}\} -^u \{\{A,\overline{A}\}\} \\
= \ & \{\{B\},\{C\},\{A\}\}
\end{aligned}$$

Note that $\{A\}$ has been added to the basic ATMS label of d by the extended ATMS computation. This means that $\mathcal{F} \cup \mathcal{T} \cup \{A\} \models d$. We note that this follows from the fact that $\mathcal{F} \cup \mathcal{T} \cup \{A\} \models B \vee C$ (since $\overline{A} \vee B \vee C$ is true), and that $\mathcal{F} \cup (B \vee C) \models d$ (since $B \Rightarrow d$ and $C \Rightarrow d$).

The computation of the label $L_{\mathcal{F}\cup\mathcal{T}}(a)$, for any atom $a \in \mathcal{L}$ is essentially the enumeration of all minimal "models" (restricted to atoms in $\mathcal{A}$) of a in $\mathcal{F} \cup \mathcal{T}$. This task is known to be #P-complete [13]. Thus, in the general case, we expect to perform computation exponential in the size of $\mathcal{A}$. deKleer [3] uses hyper-resolution to incorporate disjunctions in $\mathcal{T}$ (the cause of the exponentiality) into labels computed using the Horn theory $\mathcal{F}$. Our order-theoretic reconstruction of the computation in terms of closures and anti-chains allows us to enumerate labels directly in the space of models rather than indirectly in the space of proofs. A proof-theoretic scheme for label computation is computationally more expensive than the model-theoretic approach when there are multiple proofs of a literal a in $\mathcal{F} \cup \mathcal{T}$ based on the same support set $x \in \mathcal{E}$. All of these proofs are enumerated in the course of the application of the hyper-resolution label correction rules.

The fundamental computation in our framework to incorporate an ADNF formula τ into an anti-chain S on $\mathcal{E}$.

$$\begin{aligned}
\Phi_\tau(S) &= \overset{u}{\bigcap_{t\in\tau}} Y_t \\
Y_t &= \overset{u}{\bigcup_{s\in S}} \{s - t\}
\end{aligned}$$

Table 8: Extended ATMS Label Update.

```
function correctLABEL($L_{\mathcal{F}}, \Phi^{\mathcal{T}}$) =
  let L = $\Phi_{\mathcal{T}}(L_{\mathcal{F}}(\perp))$
  in  $\lambda$(atom) =>
        if atom = $\perp$ then L
        else $\Phi_{\mathcal{T}}(L_{\mathcal{F}}$(atom) $\cup^u$ $L_{\mathcal{F}}(\perp))$ $-^u$ L
        endif
  endlet
```

If $\tau = \{t_1, \ldots, t_n\}$ and $S = \{s_1, \ldots s_m\}$, where each $t_i, s_j \in \mathcal{E}$, then a straightforward implementation of this computation requires mn set-difference operations, mn upper unions, and n upper homogeneous intersections. Any reduction in the number of operations is a win, as is any reduction in the sizes of the arguments of the upper unions and upper homogeneous intersections. We now describe a list of optimizations that can be implemented very cheaply to achieve both types of reductions.

1. *Computing changes to labels caused by the introduction of $\mathcal{T}$.* We compute changes to labels in the Φ_τ computation, rather than the entire new label. In effect, we define the new operator $\Delta\Phi_\tau$ shown below and use it to calculate the extended ATMS label.

$$\begin{aligned}
\Phi_\tau(S) &= S \cup^u \Delta\Phi_\tau(S) \\
\Delta\Phi_\tau(S) &= \overset{u}{\bigcap_{t\in\tau}} \Delta Y_t \\
\Delta Y_t &= \overset{u}{\bigcup_{s\in S}} \text{ if } \{s-t\} = \{s\} \text{ then } \emptyset \text{ else } \{s-t\}
\end{aligned}$$

   For each s, t pair for which $s - t = s$, we save one upper union operation. In addition, the sizes of arguments to the remaining upper unions is reduced, since we only work with ΔY_t's rather than the Y_t's themselves. This optimization, shown in Table 9, employs the same intuition as that used by deKleer and Forbus in their basic ATMS algorithm shown in Table 6.

2. *Detection of early termination.* During the computation of $\Phi_\tau(S)$ we can detect conditions under which $\Phi_\tau(S) = S$, so that we can

Table 9: The optimized $\Phi_\tau(S)$ computation.

```
function $\Phi_\tau$(S) =
  let $\Delta\Phi_\tau$(S) = $\{\emptyset\}$
  in for $t$ $\in$ $\tau$ do
       let $\Delta Y_t$ = $\emptyset$
       in for $s$ $\in$ S do
            if $s - t \neq s$ then $\Delta Y_t$ = $\Delta Y_t$ $\cup^u$ $\{s - t\}$ endif
          endfor
      $\Delta\Phi_\tau$(S) = $\Delta\Phi_\tau$(S) $\cap^u$ $\Delta Y_t$
     endfor
  return S $\cup^u$ $\Delta\Phi_\tau$(S)
  endlet
```

terminate the computation early. Suppose there is a $t \in \tau$ such that for all $s \in S$, $s - t = s$. It is easy to see that the corresponding $Y_t = S$. We have $[\cap^u Y_t] \preceq Y_t$ by the property of $\cap^u$. But $\Phi_\tau(S) = \cap^u Y_t \supseteq S$ since Φ_τ is a closure operator. Therefore, whenever $Y_t = S$ for some $t \in \tau$ then $\Phi_\tau(S) = S$, and we can stop the label computation. At best, we save $(m-1)n$ set differences, $(m-1)n$ upper unions and n upper homogeneous intersections. At worst, i.e., when this condition is true of the last $t \in \tau$ examined, we save n upper homogeneous intersections.

3. *Simplifications involving* $\cup^u$ *and* $\cap^u$. We use properties of $\cap^u$ and $\cup^u$ to simplify the computation of individual Y_t's as well as $\Phi_\tau(S)$.

$$\begin{aligned}\{\{\emptyset\}\} \cup^u S &= \{\{\emptyset\}\} \text{ for any } S \in \text{Anti}(\mathcal{E}) \\ \{\{\emptyset\}\} \cap^u S &= S \text{ for any } S \in \text{Anti}(\mathcal{E})\end{aligned}$$

Thus, for instance, if we encounter an s, t pair in the computation of a specific Y_t, such that $s \subseteq t$, then $s - t = \emptyset$ and we can immediately report $Y_t = \{\{\emptyset\}\}$ saving upto $m - 1$ upper unions and 1 upper intersection in the computation of $\Phi_T(S)$.

6 Acknowledgements.

This project would not have come into existence without the participation of Prakash Panangaden, who played a key role in inciting and helping us to explore the potential usefulness in AI representations of the kinds of order-theoretic structures being used in the semantics of concurrency. He is also responsible for showing the key role that closures play in the ATMS. Haym Hirsh was helpful to us in understanding the usefulness of abstractions for the VS algorithm. We also thank anonymous referees who have read the paper as well as Ziqiang He, Leonid Libkin, Rona Machlin, and Bonnie Webber. Gunter's work was supported in part by ONR grant N00014-95-1-0245. Ngair's work was supported by the Institute of Systems Science, Singapore. Subramanian's work was supported by NSF grant IRI-8902721.

A Anti-Chain Library Interface

In this appendix we describe signatures for a Standard Meta-Language implementation of anti-chains over lattices. With the brief explanation we now provide, these should make sense to readers not familiar with SML. An SML *signature* is a list of names expected to be present in an implementation of the signature; such implementations are called *structures.* Signatures contain the names of structures, types, exceptions, and values; the cases we consider in this section contain only names of types, exceptions, and values. In a signature, a value name is given together with its type. For example, to declare that the name `singleton` denotes a value mapping `elt`'s to `ac`'s, one includes the line

```
val singleton : elt -> ac
```

in the signature. The signature for anti-chains is given at the end of the paper. It indicates, for example, that an anti-chain structure contains a type called `elt` and a type called `ac`. There is also an *exception* called `NotFound` which is used to signal the failure to find an `elt` in an `ac` and a value `empty` which is of type `ac`. The signature does not describe the semantics of these objects. For instance, it does not say that `elt`'s will be viewed as elements of `ac`. It describes only the types of the values, and lists the names of exceptions and types that are present.

Before giving a more detailed discussion of the particular values in the `ANTICHAIN` signature, we sketch the role that this signature plays in programming with anti-chains. Anti-chains are special kinds of subsets of a poset, so it makes no sense simply to speak of anti-chains independently of the posets over which anti-chains are being taken. The signature `ANTICHAIN` should be viewed as part of the 'type' of an operator from environments (sets of bindings) to environments, taking as its parameter an environment defining a lattice is defined and producing a new environment in which operations on anti-chains over that lattice are provided. Like anti-chains, lattices come with a collection of operators we expect to be present; they are given as a signature `LATTICE` in Table 10. The particulars of this signature will be discussed shortly. Now, an implementation of anti-chains is an operator called a *functor* (the SML term for a parameterized structure) that takes a lattice structure (that is, an implementation of the signature `LATTICE`) and produces a anti-chain structure (that is, an implementation of the signature `ANTICHAIN`) over that lattice structure. If we analogize with functions and types we might write this as follows:

```
functor Lattice2AC : LATTICE -> ANTICHAIN
```

Table 10: LATTICE

```
signature LATTICE =
  sig
    type elt
    datatype relationship = Less | Greater | Equal
    datatype 'a option = Some of 'a | None
    val latOrd : elt * elt -> relationship option
    val bottom : elt
    val meet : elt * elt -> elt
    val top : elt
    val join : elt * elt -> elt
    val sortOrd : elt * elt -> relationship
  end (* LATTICE *)
```

In summary, the implementation of anti-chains is given by the coding of a transformation such as this.

Let us begin by discussing the signature `LATTICE` for lattices given in Table 10. According to its definition, a lattice is a set together with a relation, two constants, and two binary operators. In `LATTICE`, the lattice elements are drawn from a type called `elt` and the order relation `latOrd` on `elt`'s is represented as a function mapping pairs of `elt`'s to values of a type `relationship option`. The values of `relationship` represent $\prec$ (`Less`), $\succ$ (`Greater`), and $=$ (`Equal`). In a lattice a given pair of elements may satisfy none of these relationships, so the `latOrd` operation takes a pair of lattice elements and produces a `relationship` *option* as its output. An element of `relationship option` either has the form `Some x` where `x` is a `relationship` or has the form `None`. The constants are `bottom`, `top` and the binary operations are `meet`, `join`.

The function `sortOrd` is not part of the the mathematical definition of a lattice. It is given here for purposes of efficiency and its significance arises when we wish to form sets (or anti-chains) of lattice elements. To represent sets efficiently, it is often useful to have a linear ordering of set elements (that is, an ordering in which any two elements are related). This allows sets to be represented as balanced trees so that searching for an element can be done quickly. It is important to appreciate that the ordering used for such balanced trees generally *must* be different from the ordering `latOrd` on the lattice since a lattice need not be a linear ordering (as reflected in the fact that the image of `latOrd` is a relationship *option* rather than a relationship). Note in particular that if we are representing anti-chains relative to the lattice ordering then there will be *no* relationship between pairs of elements of the anti-chain!

The semantics of the signature `LATTICE` is given by the mathematical lattice axioms together with the stipulation that `sortOrd` is a linear order. An implementation of `LATTICE` is assumed to satisfy this semantics, although SML cannot check that it does.

Now let us turn to the signature `ANTICHAIN` which is given in Tables 11 and 12. The semantics of most of the interface operations in Table 12 of the signature are described by the mathematics in Section 2 (assuming a self-evident mapping of the names). Constants and operations in the Table 11 part are taken by analogy with other operations in the sets signature of the SML/NJ library. The semantics of these are described succinctly in notes delimited by the comment characters `(*` and `*)`. The values described in the first half of `ANTICHAIN` are ones that are basically the same for both sets and anti-chains or that apply only to anti-chains. The important thing to note is whether an operation refers to the anti-chain or to the downward- or upward-closed set that the anti-chain is meant to represent. This distinction means nothing for functions like `singleton` and `equal` which are the same regardless of which meaning is taken. However, it is essential to note that the function `numElts` gives the number of elements in the representing anti-chain rather than the number of elements in a lower or upper set represented by it. In the semantic description given as comments in the signatures, this distinction is made by distinguishing consistently between the *anti-chain* and the *set*. So, for instance, the comment

```
(* Return the number of elt's in the anti-chain. *)
```

means that `numElts` takes a set represented as an anti-chain as an argument and returns the number of elements in the representing anti-chain. If we had wanted to know how many elements were in the set that the anti-chain is meant to represent, we would need to know whether the anti-chain represents its upper set or its lower set (and have some way of enumerating its elements or otherwise counting them).

For the functions `app`, `revapp`, `fold`, and `revfold`, the order that is *increasing* or *decreasing* must, of course, be `sortOrd`.

The values declared in the second column of `ANTICHAIN` come in two flavors, `upper_` or `lower_` depending on whether the anti-chain is viewed as representing an upper set or a lower set. So, for instance, the application `upper_add(S,x)` inserts `x` into the upper set `S`. If `S'` is the anti-chain representing `S` then this means that `x` is added to `S'` unless there is an element `y` of `S'` such that the value of `latOrd(x,y)` is not `Greater` or

`Equal`, in which case the value is simply `S'`. If we had applied `lower_add` instead, we would check whether `sortOrd(x,y)` is not `Less` or `Equal`. Similarly, the functions for `find`'ing and `peek`'ing do their finding and peeking in upper or lower sets depending on how they are prefixed.

The key points about these interfaces and the way they have been described are these:

1. The semantics of the interfaces are given abstractly so that the mathematical model is clear and does not over-constrain the implementation.

2. What to include in the interface was based on a selection of the mathematical primitives needed to express the algorithms which the implementations of the interfaces are intended to support.

3. The interface language in which the sets of operations are described provides types and abstractions supporting a substantial but computationally feasible part of the task.

Although an emphasis on mathematical and implementation independent descriptions is desirable, the choice of interfaces will be significantly influenced by a tension between available implementation techniques and the kind of reuse that the programmer is trying to achieve. The interfaces provide a vocabulary in which to discuss these trade-offs more formally. Let us illustrate. In some contexts using `Lattice2AC` may prove awkward or inefficient. For example, although the mathematics of lattices calls for top and bottom elements, it is possible to implement `ANTICHAIN` without using them. Moreover, in some cases where one has a lattice mathematically, one or the other of these elements may be difficult to implement. Hence it is often desirable to use a 'thinner' lattice signature, `LATTICE'` that omits `top` and `bottom`. In SML an implementation of `LATTICE'` is still an implementation of `LATTICE`, so little is lost by this thinning.

Another serious issue arises when one knows something about the input lattice that can be useful in the efficient implementation of anti-chains over it. For instance, if one knows that the lattice will be a boolean lattice over a finite set of atoms, then the anti-chain implementation may optimized by taking advantage of this fact. A functor from `LATTICE` or `LATTICE'` to `ANTICHAIN` cannot do this because its input interface lacks the needed primitives. Moreover, it is quite simple to describe boolean lattices because all one needs to know are the atoms; the lattice operations can all be defined in terms of whatever representation of sets of atoms one chooses to use. For these two reasons it probably makes more sense to organize code into a functor that takes an 'atoms' model as its input. So, given a signature like

```
signature ATOMS =
  sig
    type elt
    val eq : elt * elt -> bool
    val atoms = elt list
  end (* ATOMS *)
```

one implements

```
  functor Atoms2AC : ATOMS -> ANTICHAIN
```

Whether anti-chains over a lattice are produced using `Lattice2AC` or `Atoms2AC`, the mathematical semantics of the anti-chain operations should remain the same. The implementations will undoubtably differ.

Table 11: ANTICHAIN

```
signature ANTICHAIN =
  sig
    type elt
    type ac
    exception NotFound

    val empty : ac
        (* Empty ac *)

    val singleton : elt -> ac
        (* Create a singleton ac *)

    val isEmpty : ac -> bool
        (* Return true if and only if the ac is empty. *)

    val equal : (ac * ac) -> bool
        (* Return true iff the two ac's are equal *)

    val numElts : ac ->  int
        (* Return the number of elt's in the ac *)

    val listElts : ac -> elt list
        (* Return a list of the elt's in the ac *)

    val app : (elt -> 'b) -> ac -> unit
        (* Apply a function to the elt's in the
         * ac in decreasing order *)

    val revapp : (elt -> 'b) -> ac -> unit
        (* Apply a function to the elt's in the
         * ac in increasing order *)

    val fold : (elt * 'b -> 'b) -> ac -> 'b -> 'b
        (* Apply a folding function to the elt's
         * in the ac in decreasing order *)

    val revfold : (elt * 'b -> 'b) -> ac -> 'b -> 'b
        (* Apply a folding function to the elt's
         * in the ac in increasing order *)

    val exists : (elt -> bool) -> ac -> elt option
        (* Return an elt in the ac satisfying the predicate
         * if any, return NONE if there is none *)
```

Table 12: ANTICHAIN (continued)

```
    val upper_add : ac * elt -> ac
    val lower_add : ac * elt -> ac
        (* Insert an elt *)

    val upper_find : ac * elt -> elt
    val lower_find : ac * elt -> elt
        (* Find an elt in an set, raise NotFound
         * if not found *)

    val upper_peek : ac * elt -> elt option
    val lower_peek : ac * elt -> elt option
        (* Look for an elt in a set, return NONE
         * if the elt is not there. *)

    val upper_member : ac * elt -> bool
    val lower_member : ac * elt -> bool
        (* Return true iff elt is in the set *)

    val upper_subset : (ac * ac) -> bool
    val lower_subset : (ac * ac) -> bool
        (* Subsets *)

    val upper_difference : ac * ac -> ac
    val lower_difference : ac * ac -> ac
        (* Difference. *)

    val upper_union : ac * ac -> ac
    val lower_union : ac * ac -> ac
        (* Union *)

    val upper_homogeneous_intersection : ac * ac -> ac
    val lower_homogeneous_intersection : ac * ac -> ac
        (* Homogeneous intersection. *)

    val upper_heterogeneous_intersection : ac * ac -> ac
    val lower_heterogeneous_intersection : ac * ac -> ac
        (* Heterogeneous intersection. *)

  end (* ANTICHAIN *)
end
```

References

[1] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[2] Johan de Kleer. An assumption-based TMS. *Artificial Intelligence*, **28**:127–162, 1986.

[3] Johan de Kleer. Extending the ATMS. *Artificial Intelligence*, **28**:163–196, 1986.

[4] K. D. Forbus and J. de Kleer. *Building Problem Solvers*. MIT Press, 1993.

[5] Yasushi Fujiwara, Yumiko Mizushima, and Shinichi Honiden. On logical foundations of the ATMS. *Proc. of ECAI-90 Workshop on Truth Maintenance Systems*, 1990.

[6] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. The MIT Press, 1992.

[7] Haym Hirsh. *Incremental Version Space Merging: A General Framework for Concept Learning*. Kluwer Academic Publishers, 1990.

[8] C. Mellish. The description identification problem. *Artificial Intelligence*, 52:151–167, 1991.

[9] T. Mitchell. The need for biases in generalization. In J. Shavlik and T. Dietterich, editors, *Readings in Machine Learning*. Morgan Kaufmann, 1990.

[10] Tom Mitchell. *Version Space: An approach to Concept Leaning*. PhD thesis, Stanford University, 1978.

[11] T-H. Ngair. *Convex Spaces as an Order-Theoretic Basis for Problem Solving*. PhD thesis, University of Pennsylvania, 1992.

[12] D. S. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In J. Fox, editor, *Computers and Automata*, pages 19–46. Polytechnic Institute of Brooklyn Press, 1971.

[13] J. Simon. *On some central problems in computational complexity*. PhD thesis, Cornell University, 1975.

[14] M. Smyth. The largest cartesian closed category of domains. *Theoretical Computer Science*, **27**:109–119, 1983.

[15] S. Vickers. *Topology via Logic*, volume 5 of *Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.